HDswitch
An interactive live video mixer for HD videos

Stephan Soller, Computer Science and Media
Stuttgart Media University
ss312@hdm-stuttgart.de

Abstract

Video streaming has become common place to broadcast and record conferences and university lectures. More complex video streaming setups combine multiple video and audio sources into one live stream users can watch over the internet. Usually a human operator does this with some kind of video and audio mixer. This paper describes the motivation for building a new video audio mixer application, it's requirements, design as well as some details of its implementation.

1. Introduction

Simple video streaming scenarios only use one video and audio source (e.g. a webcam). Such a stream can be broadcasted directly by a streaming server like Icecast [1] or send to a streaming platform like Ustream [2]. IP cams even integrate a streaming server and can broadcast directly without additional hardware or software.

More complex setups combine multiple video and audio sources. For example one camera showing the speaker, one frame grabber to capture slides and multiple microphones (speaker, presenter, questions from the audience). A human operator then uses a video and audio mixer to combine these sources into one video stream. Such a setup is often necessary for lager university lectures and conference talks.

There are already many video mixer applications targeting professional studio broadcasting (Wirecast [3]), conference live streaming (DVswitch [4]) and Let's Play sessions (Open Broadcaster Software, short OBS [5]). Most of these solutions are often designed as monolithic "one size fits all" applications (e.g. Wirecast, OBS). While quite feature rich this makes it difficult to add additional automatic or manual processing stages (e.g. automatic cutting and high quality archiving). The rest of the streaming system then grows around those limitations. So in the end the system is more defined by workarounds than by the actually necessary data flows and actions.

The monolithic nature also makes error handling quite challenging. Hardware and software errors in other parts of the streaming system can often be recovered from (e.g. restarting a streaming server or reinitializing a camera), but most video mixers don't allow to define complex error handling. In the worst case a simple error leads to a zombie system that still looks fine to the operator but neither streams nor archives any data. With much effort it's often possible to build the system so that errors lead to a complete collapse of the entire system. This is then noticed by the operator who can then restart it. But this is far away from proper error recovery.

Many features also lead to more complex user interfaces and interactions. This requires to educate operators and makes the software more difficult to use, especially in stressful live situations.

Other video mixers simply can't handle HD video. DVswitch for example is limited to the standard definition DV video format. Others can handle HD video in theory but fail to do so because of performance problems. In case of live video just one element (e.g. a color space conversation) in the entire streaming pipeline has to fail or be to slow to ruin an entire system. Debugging and performance tuning of these problems can often take weeks. Even after the cause is determined it's often impossible to properly solve these problems because the used software can't be changed accordingly.

Experience gained with two older live streaming systems (one based on Wirecast, one on DVswitch) lead to the desire to tackle those problems at their roots. Designing a new video mixer application from the ground up opens many possibilities:

- All required features can be integrated properly without leading to workarounds in other parts of the system.

- Unreliable or performance hungry components can be sidestepped where possible or processed by the proper hardware (CPU, GPU or hardware decoder or encoder).
- The user interface can be kept as simple as possible.

2. Requirements

HDswitch tries to meet the following requirements. These are not strict requirements for the first prototype but HDswitch should meet them once it's fully implemented. So they are the base for all design decisions:

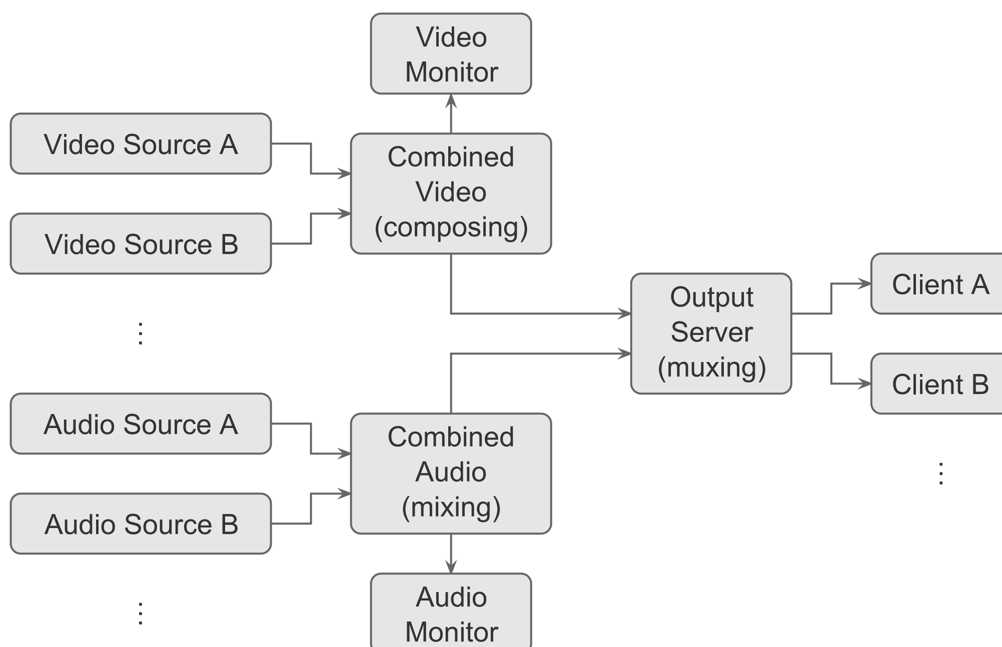
- Composite at least two 1080p video sources at 60 FPS into one output video stream.
- Composite at least two microphone sources into one output sound stream.
- Fast switching between configured scenes (e.g. show only stream A, only stream B or picture-in-picture with A and B).
- Allow the operator to get realtime feedback of the final video and audio output.
- Make it easy for the operator to signal the start and end of individual talks in the stream (e.g. by pressing the space key) and enter the meta data for a talk (e.g. title and speaker).
- Easy, clean and loseless machine interface for further processing of the output video.
- Generate a correct output video with proper timestamps so that later processing stages don't endanger audio and video synchronization.
- Allow to implement automatic error recovery on as many levels as possible.

3. Basic design

Figure 1 shows the basic structure of how HDswitch processes video and audio data:

- The video streams from multiple video sources are composed into one combined video. Either by showing only one of the source streams or by combining them ala picture-in-picture.
- The final video is then displayed for the operator (video monitor) and send to the output server.
- Basically the same happens for the audio sources: All sources are combined into one audio stream (mixing). This stream is then played back for the operator (audio monitor) and send to the output server.
- The output server combines all incoming audio and video data into one data stream and distributes it to all interested clients.
- Clients are other programs implementing later processing stages, e.g. an ffmpeg process that encodes the video into WebM and sends it to a live streaming server for broadcasting.

Figure 1: Dataflow and structure of HDswitch video and audio processing



The APIs used by HDswitch are primarily selected to limit 3rd party error sources and cut down external dependencies to readily available APIs.

The Video 4 Linux 2 (V4L2) API [6] is used to interact with USB webcams and USB 3.0 frame grabbers. Or generally speaking everything that implements the USB Video Class (UVC) protocol. V4L2 is the direct Kernel level interface for this purpose. As such it it well documented and reliable. Libraries or frameworks would also build upon V4L2 and using the API directly eliminates abstraction or programming errors in those components.

The ALSA API [7], which is the kernel level audio interface, was initially used to capture audio data. But management of many audio sources with ALSA is quite error prone and CPU intensive. Most standard Linux distributions today ship with the PulseAudio sound system [8]. And those distributions emulate the ALSA API via PulseAudio. So even when using the ALSA API the data would pass though PulseAudio. Therefore PulseAudio was directly used to capture audio data. This removed much of the CPU time and PulseAudio offers advanced latency control and easy management of multiple audio sources.

All image processing is done on the GPU via OpenGL [9] (e.g. resizing, blending, composing, color space conversation). The open source graphics driver and OpenGL stack is stable and fast enough so that even cheap integrated GPUs have no trouble with the necessary image processing. The stack also uses the same OpenGL implementation for GPUs of different vendors (AMD, nVidia and Intel). Device drivers are handled in vendor specific backends. This limits vendor specific differences in the OpenGL API as most code is used for all vendors. The open source graphics stack is also part of the Linux kernel and therefore available out of the box without any driver installation or configuration. By focusing on this OpenGL stack and a relatively conservative set of OpenGL extensions HDswitch should run on any recent Linux kernel of the last few years.

HDswitch doesn't use any GUI library (e.g. GTK or KDE) as such libraries are difficult to integrate into an event based mainloop. They also add quite a lot of dependencies and error sources. Instead the little bits of GUI HDswitch actually needs are rendered directly via OpenGL. This also guarantees that the video preview can be rendered properly. Thats important because standard GUI libraries sometimes add CPU rescalers or color space conversations to display a video (depending on system configuration, version or other installed software and updates). In case of HD video this can easily eat up most of the CPU time, killing the performance needed for HD material.

To create a desktop window and establish an OpenGL context the SDL library v2.0 [10] is used. It's a simple and relatively lightweight library that doesn't impose any specific architecture on the rest of the program.

UNIX domain sockets [11] are used to transfer the output video to clients. UNIX domain sockets are optimized for local inter process communication and are easily fast enough for 1080p video material at 60 Hz (about 240 MiByte/s). The output data stream is a normal Matroska video [12] with uncompressed video and audio data. Additional events and data can be easily embedded into the Matroska video as a separate data stream. This is very useful to signal the start and end of individual talks as well as to transmit the meta data of such talks. Since every data packet in Matroska already has a timestamp this approach allows for exact and easy automatic video cutting in later processing stages. If the data is not needed the extra data track can simply be ignored or deleted.

These choices have been made so clients don't need to implement an extra protocol to get data from HDswitch. UNIX domain sockets can be read directly with standard video processing tools like ffmpeg and Matroska is a well supported container format. The output data can also be easily used in any form of program or shell script to add more complex processing. Thanks to this HDswitch can be easily combined with the full power of shell scripting (e.g. gzip the output stream and dump it to disk or send it over a high speed network). The socket API is also easy to integrate into a non-blocking event loop.

An alternative to this would be to use shared memory buffers. But clients would need to implement this protocol and transmission of additional data streams would be more difficult. Since UNIX domain sockets are fast enough for the application at hand there is no need to increase complexity.

Video encoding is kept out of HDswitch. It only does video and audio mixing and tries to change the input material as little as possible. Later stages (e.g. an ffmpeg client process) can then take care of video encoding. Since video encoding is easily the most CPU intensive processing step it's often the focus of prolonged performance and quality tuning. Standard tools like ffmpeg

have a multitude of options for these purposes and if HDswitch would include an encoder it would have to provide all these options as well.

Not encoding the output video also allows later stages to efficiently do post processing on the raw material. E.g. an ffmpeg process can read the raw video from the UNIX domain socket and apply a nose filter to the video. If HDswitch would already encode the output video noise would not be encoded as noise but as encoding artifacts. This would prevent any later processing stages from applying a proper nose filter.

4. Implementation

HDswitch avoids computationally intensive processing, handles it with the GPU or lets external processes take care of it (e.g. an ffmpeg process takes care of video encoding). What remains is mainly coordination of how the data streams are moved between CPU, GPU and clients. HDswitch doesn't offer much parallelism in the sense of CPU bound work that should be distributed over multiple CPU cores. Instead it mostly requires I/O multiplexing. HDswitch takes advantage of that by using a single threaded `poll()` [13] based event loop and non-blocking I/O for its program logic. This makes event based programming the main paradigm.

Thanks to the universal I/O principle of UNIX and some Linux specific system calls events from almost every kind of source can be handled as a file descriptor (basically a handle to a kernel object).

- The Video 4 Linux 2 API already uses file descriptors, one for each opened device. If such a file descriptor is readable a new video frame is ready on that device.
- The socket API also uses file descriptors. If the server file descriptor is readable a new client connection is ready and can be established by calling `connect()`. That system call in turn returns new file descriptors for each client connection. If such a connection file descriptor is writable the client is ready to receive more data.
- `timerfd_create()` [14] can be used to create file descriptors that are readable after a specified time or in specified intervals. HDswitch uses one to periodically ask the SDL library for pending mouse, keyboard and window events (unfortunately SDL doesn't expose a file descriptor to wait for it's events).
- `signalfd()` [15] provides a file descriptor to handle incoming UNIX process signals (e.g. SIGINT and SIGTERM). If the file descriptor is readable a new signal is pending and can be processed.

All these different file descriptors can then be monitored with one looped `poll()` system call. This system calls basically returns a list of all file descriptors ready for further processing or waits until one is ready. If for example a Video 4 Linux 2 file descriptor is reported to be readable HDswitch calls the code to read all pending frames from the device.

Threads would offer an alternative to implement this kind of I/O and event multiplexing. It would however require difficult and error prone thread synchronization. The influence of latency control and GUI interactions on other threads would be quite complex leading to a high risk of deadlocks. A single threaded environment also makes it easier to experiment with complex control flow and event causality. This was very useful when exploring latency control implementations.

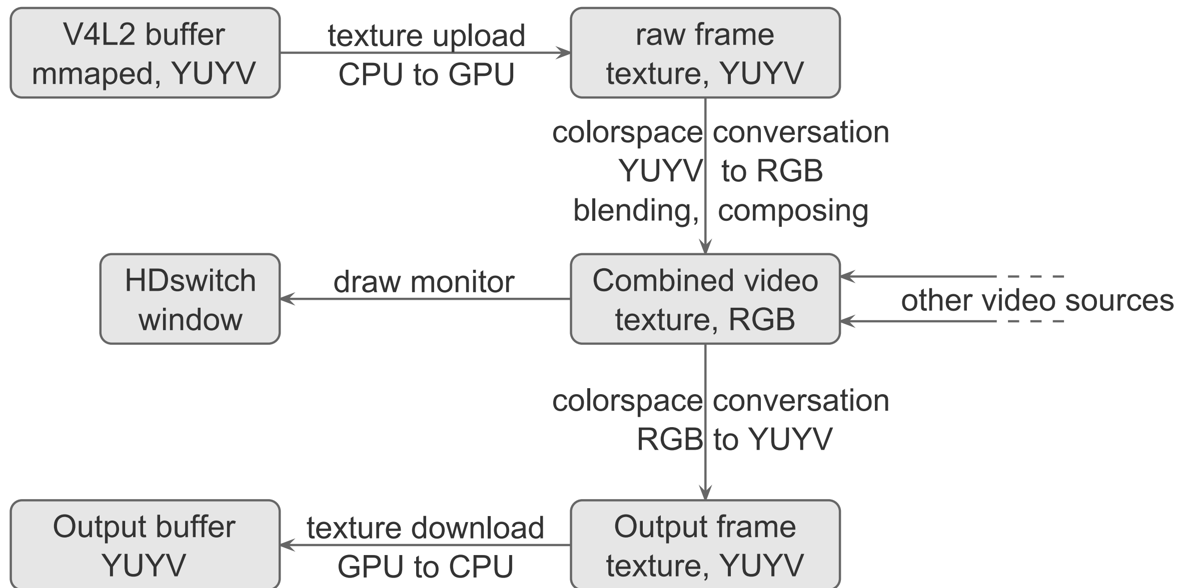
Unfortunately there is one major API that can't be easily integrated into a `poll()` based mainloop: PulseAudio. PulseAudio itself uses its own `poll()` based mainloop and provides an interface to make PulseAudio use a custom mainloop. So it's possible to teach PulseAudio how to use the mainloop of HDswitch. But this would require much more code than seemed reasonable at the time. After a short discussion on the PulseAudio mailing list a different approach was chosen: PulseAudio more or less provides the same functionality than the mainloop implemented for HDswitch. So the entire event handling code of HDswitch was migrated to use the PulseAudio event loop. The custom event loop implementation used previously was discarded. This allowed to use the event based programming paradigm for everything and eliminated some code redundancy while maintaining the same functionality.

Efficient video stream handling

Since HDswitch should process 1080p video streams with 60 Hz efficient handling of video frames is important. Figure 2 shows how this process is implemented by HDswitch.

The Video 4 Linux 2 API offers to directly map video buffers into process address space. Those buffers are filled with data by the USB device and the kernel then notifies HDswitch that the buffer is ready. One buffer

Figure 2: Video processing steps taken by HDswitch.



then contains a complete video frame in a raw format (YUYV, or more precisely YCrCb with 4:2:0 subsampling).

The complete video frame is then copied to an OpenGL texture as it is. No processing occurs on the CPU. HDswitch statically allocates textures for all streams and overwrites the same texture memory for each new frame (using the `GL_ARB_texture_storage` extension). This is different from the default OpenGL procedure that would create a new texture for each frame. Statically allocating and overwriting a texture for each stream avoids fragmentation of GPU memory which in turn could cause fluctuations in the OpenGL performance.

Each time a new frame is available the composite frame is redrawn. This frame contains all frames that make up the current output video. For example first the latest video frame of video source A, then the latest video frame of video source B scaled down to the lower right corner (picture-in-picture). During this drawing operation a fragment shader performs the color space conversation from 4:2:0 YUYV to RGB as well as any required alpha blending. Scaling is automatically taken care of by the GPU texture interpolation.

The resulting composite frame is then drawn into the HDswitch window to give the operator feedback of what the output currently looks like. After that the composite frame is again converted to 4:2:0 YUYV by using another fragment shader. The result is copied back to CPU memory and handed over to the UNIX domain socket server which wraps it into an Matroska container and distributes it to all connected clients.

This scheme creates n+2 drawcalls per incoming frame where n is the number of video sources that are visible on the composite frame. The frame data is copied at least once (from the V4L2 buffer into an internal OpenGL buffer). Additional copies are created when the output frame is distributed over the UNIX domain sockets.

The drawing code and texture handling are currently implemented in a straight forward manner and still offer many performance optimizations (e.g. OpenGL pixel buffer objects and shader optimizations). Same applies to the code that distributes the frames to all clients.

Latency handling and timestamp generation

Latency is an important factor in live streaming systems. HDswitch combines data from possibly many different microphones, webcams and frame grabbers. And each video frame and piece of audio can have a different latency. That is the time it takes from its recording on the device until HDswitch finally gets the data via PulseAudio or Video 4 Linux 2. But in the output video all sources should be properly synchronized. HDswitch achieves this either by configuring and monitoring the latency of the sources (audio) or by trying to keep the latency as low as possible (video).

HDswitch requests a 10 ms latency from PulseAudio and continuously monitors the actual latency reported by PulseAudio. On the few tested development systems

this lead to an reported audio input latency of 0.5 ms to 2.5 ms. To keep video latency as low as possible HDswitch only uses raw (YUYV) data from webcams and frame grabbers. Using MJPEG or H.264 added noticeable delay to the video input.

The latency information from PulseAudio is also used to properly synchronize and mix multiple audio inputs and calculate proper timestamps for the output audio. Since Video 4 Linux 2 does not seem to delay input frames HDswitch doesn't pay attention to video input latency.

The timestamps of the output video tell encoder programs and media players when a part of video or audio data should be displayed or played. Video players use that information to properly synchronize audio and video playback. But media players also do a lot of error correction and guessing to properly play videos with incorrect timestamps. Encoder programs (e.g. ffmpeg) usually don't do this kind of error correction but try to keep the video as original as possible. This makes it possible that a video with incorrect timestamps plays find on most media players but breaks once encoded to another format (e.g. from MP4 to WebM). Audio video synchronization can be off, can drift slowly away while the user watches the video or seeking can be broken.

To avoid such effect in later processing stages HDswitch takes care to write correct timestamps. This is done by measuring the timestamps as soon as possible and by including all available latency data.

The second aspect of HDswitch strongly influenced by latency is the monitor output. Thats the realtime output of the final video also distributed to the clients. So the operator gets a proper preview of what viewers on the Internet would see.

The human operator wants to see and hear the output video as soon as possible. Large delays or asynchronous audio (e.g. 1 or 2 sec. delay) can strain the operator quite heavily (e.g by causing slight disorientation or headaches). Making it very demanding if not impossible to work for 8 hours.

To keep the latency to the monitor output as low as possible HDswitch shows or plays the final video as soon as it is created (see figure 1). This however leads a pretty fast but more or less unsynchronized video and audio monitor output. The video monitor is updated directly after a new frame is received. But new audio data is kept back until the mixer has data from all audio sources. If the latency of one source exceeds 40 ms the mixer no longer waits for that source (at 40 ms latency starts to be noticeable). The idea is to allow the operator to actually hear which device causes the large latency and fix it.

5. Limitations and further work

HDswitch can composite two 1080p streams at 60 fps but isn't able to send such a output video to clients. Not enough time is spend in the output server code that actually sends the data to the clients (more specifically in the `write()` system call). The cause is not yet clear as CPU utilization is at about 70%. Possible causes are stalls in the OpenGL pipeline or some unknown behavior of the PulseAudio mainloop. Until now all testing was performed with unoptimized debug builds so a short term solution might be to enable compiler optimizations (e.g. `-O2`). A proper solution would probably be to extract the output server into it's own thread or use AIO (which glib implements as an extra thread).

During development HDswitch was mainly test with cheap consumer hardware: analog microphones for about 10€ and webcams in the 50€ to 100€ range. All this hardware showed reasonable (unnoticeable) latency. One configuration HDswitch was developed for should use a prosumer camera (about 1500€) and an HDMI to USB 3.0 frame grabber (about 400€). PulseAudio reports an audio latency of about 300 ms for the camera and about 550 ms for the HDMI frame grabber. Unfortunately this doesn't seem to be a bug in PulseAudio as the same could be reproduced on multiple Windows systems.

This extremely high latency on more expensive hardware breaks the HDswitch monitor. The video is shown without noticeable delay but the audio is almost half a second delayed. If the mixer threshold is changed from 40 ms to 600 ms the output video will be properly synchronized. The large latencies are then properly included in the timestamp calculations. But the audio of the monitor output still be delayed by about 550 ms. Probably confusing and straining the operator.

Currently HDswitch outputs one frame for each input frame received. If two video sources with 60 fps each are used HDswitch will send 120 fps to the output video. This can be solved by waiting until all video sources delivered an input frame for the current output frame. However special care must be taken in regard to variable frame rates of webcams. In case of bad lighting webcams in-

crease their exposure time leading to frame rates as low as 7.5 fps.

Timestamps of the video frames are calculated by subtracting the current time from the time the video was started. So the video time stamps are based on the system clock. The audio timestamps are based on the number of samples received from PulseAudio. This should be based on the system clock as well but might also be based on the soundcard clock. The soundcard and system clocks can deviate over time, leading to slightly faster or slower audio. In case of long running videos this can lead to wrong timestamps and asynchronous video and audio playback. More research and testing needs to be done to fully understand how this effects HDswitch. Especially regarding when PulseAudio uses the soundcard or system clock.

The timestamps also don't snap to some constant frame rate (e.g. 30 fps) but instead are microsecond based. This allows small variation on the frame timings depending on the current program performance. As a result this might lead to micro stuttering on displays with high refresh rates (e.g. 90 Hz or 144 Hz). This can be solved easily by "snapping" timestamps to the framerate of the output video (e.g. 30 fps).

Pretty much all webcams offer compressed high resolution video formats (MJPEG or H.264). How these formats effect video latency and if and how this latency can be measured requires further research. But to use these formats each frame must be decoded before it can be moved to the GPU. This can be very CPU intensive. Therefore this approach is best combined with hardware decoder APIs, especially since these APIs allow to leave the output frame directly on the GPU (as an OpenGL texture).

6. Conclusion

The project was frozen after ⅔ of it's development time in favor of an Open Broadcaster Software (OBS) based solution. It was necessary to free development time for other processing stages that required extensive redesign and reimplementation to work around OBS's limitations. While this meant the return to workaround based design of the streaming system it allowed to introduce additional developers to the streaming project.

In its currently incomplete state HDswitch is working for most use cases but needs some further work until the performance and usability requirements are met.

7. References

[1] Icecast.org
http://www.icecast.org/
Retrieved 2014-07-14

[2] Ustream website
http://www.ustream.tv/
Retrieved 2014-07-23

[3] Wirecast
http://www.telestream.net/wirecast/
Retrieved 2014-07-23

[4] DVswitch
http://dvswitch.alioth.debian.org/wiki/
Retrieved 2014-07-23

[5] Open Broadcaster Software
http://obsproject.com/
Retrieved 2014-07-23

[6] Video for Linux Two API Specification
http://linuxtv.org/downloads/v4l-dvb-apis/v4l2spec.html
Retrieved 2014-07-23

[7] Advanced Linux Sound Architecture (ALSA)
http://www.alsa-project.org
Retrieved 2014-07-23

[8] PulseAudio
http://www.freedesktop.org/wiki/Software/PulseAudio/
Retrieved 2014-07-23

[9] OpenGL
http://www.opengl.org/
Retrieved 2014-07-23

[10] Simple DirectMedia Layer (SDL)
http://libsdl.org/
Retrieved 2014-07-23

[11] UNIX domain sockets man page
http://man7.org/linux/man-pages/man7/unix.7.html
Retrieved 2014-07-23

[12] Matroska Media Container
http://matroska.org/
Retrieved 2014-07-23

[13] `poll()` man page
http://man7.org/linux/man-pages/man2/poll.2.html
Retrieved 2014-07-23

[14] `timerfd_create()` man page
http://man7.org/linux/man-pages/man2/timerfd_create.2.html
Retrieved 2014-07-23

[15] `signalfd()` man page
http://man7.org/linux/man-pages/man2/signalfd.2.html
Retrieved 2014-07-23