

Playback Mpeg Transport Stream: exoIMpegTS

NDK 5.7/MPTK 2.4

Application Note, 12 December 2007

Overview

This document describes the MPTK example program for the MPEG2 transport stream demultiplexer. The example program demonstrates the use of TS Demultiplexer, MPEG decoder, and audio components in order to build a MPEG transport stream player. The transport stream demux can be used in DVB and ATSC applications. As delivered, it demonstrates use with MPEG2 only. The example program can be used to decode transport streams from a file or from the FGPI (Fast General Purpose Input) interface. The setup required to playback transport streams from the FGPI interface is also described. The last section talks about AV Sync mechanisms for TS player.

When you use `build_exe` at the command line to build the application, you get an overview of all the libraries the application needs.

1 Description

[Figure 1](#) illustrates the components and connections used in the program `exoIMpegTs`. This is a typical setup for transport stream playback

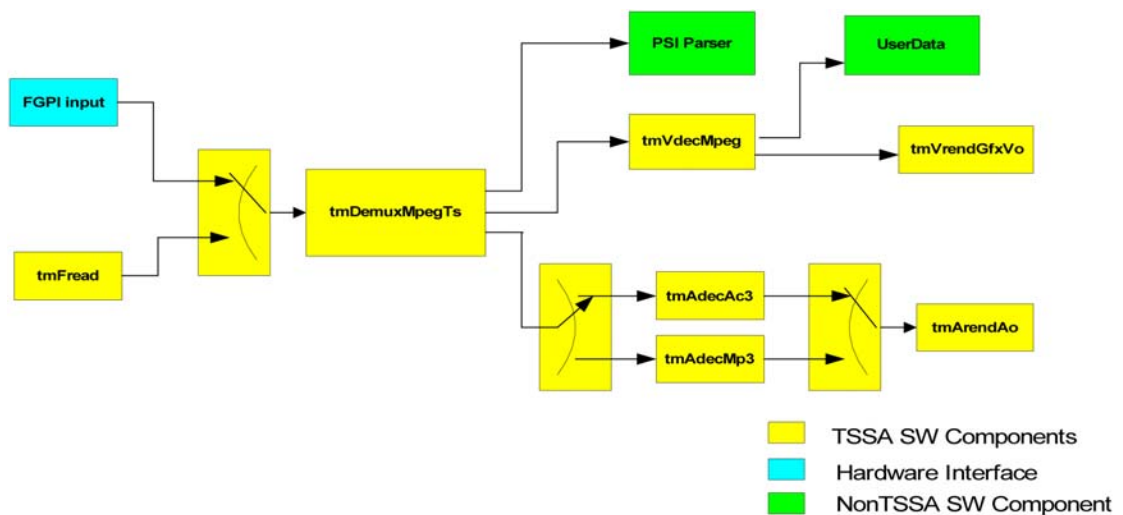


Figure 1: Transport stream player

The TS player is capable of reading the transport stream from a file present locally using the `tmFread` component. Also the transport stream can be read from a FGPI input when configured appropriately. The stream is demultiplexed by `tmDemuxMpegTs` component. The PSI (Program Specific Info) is extracted and passed on to the 'PSI Parser' to parse and output to a file. The compressed video data is fed to `tmVdecMpeg` decoder. The MPEG decoder extracts the User

Data and passes it to UserData component which extracts the relevant data and outputs to a file. The decoded video is output to video renderer. The TS player currently supports playing transport streams with AC3 or MP3 audio.

Note: The TS player does NOT currently demonstrate its use with H.264 video, AAC audio or MPEG4 video. The current ideas could however be applied to these formats as well. Also H.264 support has been demonstrated. Contact TCSHelp.com for the latest info.

1.1 Instructions

The example is built like any other MPTK example program. Follow the instructions given in other documents. It is sensitive to the `_noac3_` diversity. If you have not licensed the Dolby AC3 decoder, add this string to your `_TMTGTDIVERSITY` environment variable and then the program will be built with only the MPEG audio decoder. When you run the program, it can print this usage message:

```
Usage example: exolMpegTs.out [-infile <filename>] [fgpi] [-loopEnable][-vidpid
<pid>][-audpid <pid>] [-verboseSync][-ntsp <num>][-nmpeg2p <num>][-nac3p <num>][-nyuvp
<num>][-npsip <num>][-nuserdatap <num>] [-noVideoCarseSync][-noVideoFineSync][
noAudioCarseSync][-noAudioFineSync][-noGraceFulDegradation][-noSenderReceiverSync]
```

Playback from file:

The player can play transport streams from files present locally. To play the transport streams from a file use

```
exolMpegTs.out -infile <file> [-loopEnable] [-vidpid <pid>] [-audpid <pid>]
```

Note: To get the audio and video pids, tools like "TSReader lite" can be used. A freeware version is available from <http://www.coolstf.com>. Once installed select the source as Transport Stream file (file.dll) and then select the stream to be played. The program number and corresponding video and audio pids will be displayed on screen. Use Export->HTML Export option to save these info in html format.

Playback from FGPI:

To play the streams from FGPI input use:

```
exolMpegTs.out -fgpi
```

1.2 Start-up Options

Details of command-line options:

Command	Description/example
-h	Use this option to get help on all available options:
-infile	specifies the input transport stream file. Example: <code>-infile c:\media\oscarSD.trp -vidpid 73 -audpid 82</code>
-vidpid	The video pid of the program.
-audpid	The audio pid of the program.
-loopEnable	This is used to enable the looping of the stream. Without this the stream will play only once.
-verboseSync	This enables printing of certain messages related to AV Synchronization.

Use the next set of options to change the default number of packets between various software components.

Command	Description
-ntsp	Number of packets of transport stream data between input (Fread/FGPI) and tmDemux
-nmpeg2p	Number of packets of MPEG-2 encoded video data between demux and video decoder
-nac3p	Number of packets of encoded audio data between demux and audio decoder
-npsip	Number of packets of PSI data between demux and PSI task
-nyuvp	Number of packets of YUV video data between video decoder and renderer
-nuserdatap	Number of packet of user data between video decoder and user-data task
-npcmp	Number of packets of PCM audio data between audio decoder and renderer

Use the next set of options to change Audio Video synchronization settings

Command	Description
-noSenderReceiverSync	Sender receiver sync is disabled. The demux does not use the common clock shared between the all TSSA components of TS Player.
-noGracefulDegradation	Disables the Graceful degradation in TS Player
-noVideoCoarseSync	Disables coarse video synchronization.
-noAudioCoarseSync	Disables coarse audio synchronization.
-noVideoFineSync	Disables fine audio synchronization
-noAudioFineSync	Disables fine audio synchronization

For details on A/V Synchronization, refer to Appendix A.

Following are the default values and settings.

Default settings

(ntsp) fread	-> demuxMpegTs packets	:360	
(ntsp) fgpiWrapper	-> demuxMpegTs packets	:360	
(nac3p) demuxMpegTs	-> adec packets	:24	
(nmpeg2p) demuxMpegTs	-> vdecMpeg packets	:24	
(npcmp) adec	-> arendAo packets	:23	
(nyuvp) vdecMpeg	-> vrendGfxVo packets	:20	
SenderReceiverSync	:	:	Enabled
GracefulDegradation	:	:	Enabled
VideoCoarseSync	:	:	Enabled
AudioCoarseSync	:	:	Enabled
VideoFineSync	:	:	Enabled
AudioFineSync	:	:	Enabled

1.3 Run Time Options

The menu that appears allows the user to choose one of the following options

Menu Item	Description
1. Set Channel Using Program Number	Select the program by setting program number.
2. Set Channel Using Pid Values	Select the playback using the audio and video pids.
3. Start Time Doctor	Start Time doctor dump
4. Quit	Quit the TS player application. Stop current playback and quit.
5. Print frame drop/repeat figures	Prints player statistics like number of frame drop/holds and fine sync details.

Expected output

The introductory message should appear on the console. The decoded video should appear on the TV and (or) audio should be heard from the speakers. Some information regarding the stream should appear (the actual information that appears may vary depending on whether the stream is MPEG 1 or 2 video, MPEG audio or Dolby Ac3 audio and so on). The runtime menu is then displayed and the application prompts the user to enter a choice from the available options.

Some options of the menu have submenus or messages that prompt the user for additional information required to perform the selected option.

2 FGPI Test Setup

The setup uses DTA-102 DekTec card as transport stream generator. This is a PCI based card that can be inserted into PC. Appropriate drivers and software should be installed on that PC. This card is capable of streaming transport stream from a file on the PC in real time. The output of this card is SPI (Synchronous Parallel Interface) LVDS signal.

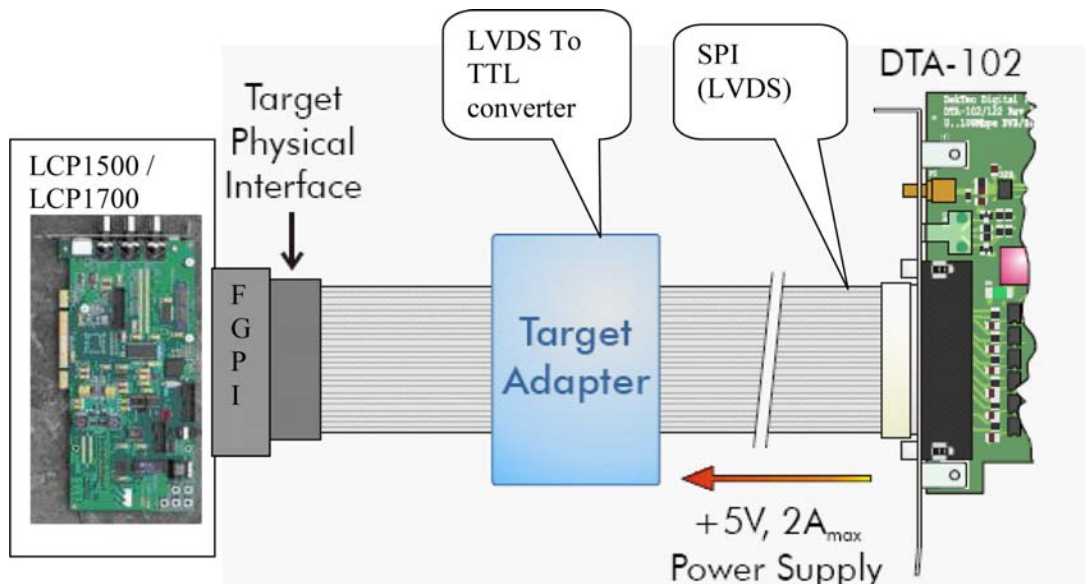


Figure 2: FGPI test setup

These signals are passed on to target adapter which is LVDS to TTL converter. The adapted output is fed to the FGPI input of LCP1500/1700 card. The exolMpegTs application should run with FGPI option enabled. Observe the output on TV. The expected output shall be as in previous section.

The details of connection from the LVDS Adapter to LCP1500/1700 are listed below.

Table 1: LVDS adapter to LCP1500/1700 connection

LVDS Adapter		FGPI Interface on PNX1500/1700	
Pin	Description	Pin	Description
1	+5V	80	+5V
16	FE_CLK	72	VDI_CLK2
17	GND	71	GND
18	FE_VALID	70	VDI_V2
19	FE_SYNC	66	VDI_D32
20	GND	1	GND
21	FE_D0	2	VDI_D0
22	FE_D1	4	VDI_D1
23	GND	5	GND
24	FE_D2	6	VDI_D2
25	FE_D3	8	VDI_D3
26	GND	9	GND
27	FE_D4	10	VDI_D4
28	FE_D5	12	VDI_D5
29	GND	13	GND
30	FE_D6	14	VDI_D6
31	FE_D7	16	VDI_D7
32	GND	17	GND

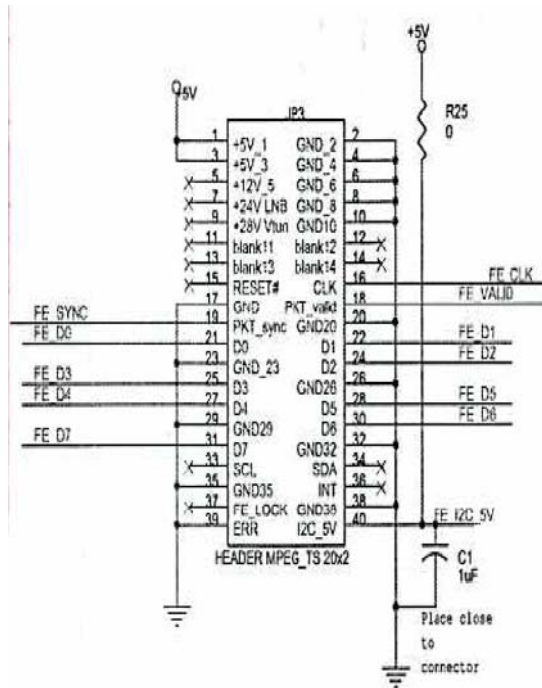


Figure 3: FGPI - LVDS pin diagram

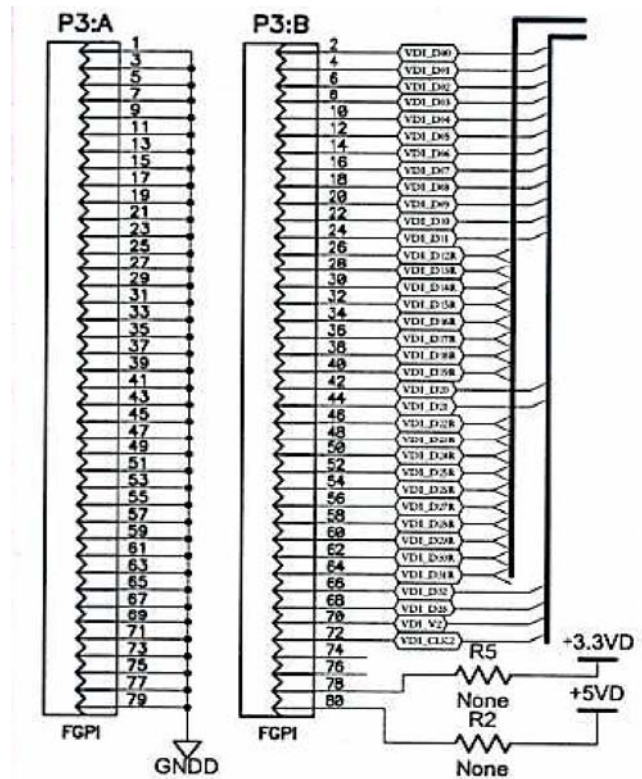


Figure 4: Pin diagram of LCP1500 FGPI 80 pin connector

Note: For connections detailing DVB-T tuner interfacing to FGPI interface please contact TCSHelp.com.

Appendix A - A/V Sync mechanism in TS Player

Overview

This section of this application note describes the implementation of A/V sync as it is done in exolDemuxMpegTs. Its effects are seen in several places. All are noted here, so that developers and system integrators stand some chance of being able to modify it. The following explanation duplicates some items that are also covered in the [Audio-Video Synchronization](#) application note. This discussion is more focused on the issues of broadcast clock recovery as implemented by the TS demux.

We can divide the discussion into three parts:

- Coarse sync describes how audio and video content are synchronized with each other and with the MPEG clock.
- Clock recovery describes how the clock is derived from the input stream.
- Fine sync describes what happens if the recovered clock differs from the 90kHz tick-rate that we expect it to have.

3 Coarse sync

Coarse sync is the process by which audio and video data can be synchronized to a given MPEG clock. This data arrives at its respective decoder in frames. Each frame requires a fixed time to appear. For video, this is typically 1/50s (NTSC) or 1/60s (PAL) for a single field. For audio, it depends on the sample rate (typically 48000 samples/s or 44100 samples/s) and the encoding standard (1152 samples/frame for MP3, 1024 samples/frame for AC3, etc.).

Each frame is accompanied by a PTS, which is the time at which it should be presented. In MPTK, the PTS is offset from the true presentation time by a value known as the sync delay. This sync delay is known at the renderer. The goal of coarse sync is to monitor the difference between the PTS of a frame (modified by the value sync delay) and the current time. If the frame has arrived early, it is held until the time arrives; if it is late, it is dropped. The difference is thus driven towards 0.

This simple process results in quick convergence between the PTSs and the clock. It results in clearly audible and visible artefacts, so video may be blanked and audio muted until convergence is reached.

3.1 MPEG clock

The MPEG clock is a reference to a clock created by `tSaClockCreate()`, with a frequency of 90kHz. This clock instance is loaded into a static structure called `gPcrClock`, of type `tSaClockHandle_t`, which is pointed to by the components that refer to it.

Note that it is typical TSSA practice to create this clock in the application. The TS demux (as of MPTK 2.1) creates this clock *INSIDE* of the demux. This is expected to be changed to match more normal usage in the future.

3.2 Audio Renderer parameters

The Audio Renderer has a completely self-contained algorithm for attaining coarse sync. In order to use it, the `tmArendAo` instance is set up with the following values in its setup structure:

defaultSetup->clockHandle	MPEG clock	
syncDelay	the sync delay, SYNC_DELAY, as defined above	
syncMode	AR_Sync_skip	
syncLockThreshold	variation between PTS and STC at which frames	
	can be dropped or held	
syncUnlockThreshold	(must be equal to syncLockThreshold)	
badTimestampThreshold	variation between PTS and STC at which PTS is	
	ignored	

The Audio Renderer estimates the presentation time of each frame and can repeat parts of frames and/or drop parts of frames as necessary. (However, this method does lead to audible clicks.) Therefore, it can quickly synchronize the stream to the MPEG clock with a tolerance of 1 or 2 ticks.

3.3 Video Renderer parameters

The Video Renderer calls its progress function to interact with the application and so achieve coarse sync. The relevant part of the progress function looks like this:

```
static tmErrorCode_t progress
(
    Int                inst,
    tsaProgressFlags_t flags,
    ptsaProgressArgs_t args
)
{
    ptmolVrendGfxVoSyncInfo_t syncInfo;
    Int                lowerLimit = ...,
                    upperLimit = ...,
                    currentDelay;

    if ((args->progressCode & VRENDGFXVO_PROGRESS_SYNC) != 0U)
    {
        syncInfo                = args->description;

        currentDelay =
            syncInfo->pcktTimeStamp +
            SYNC_DELAY -
            syncInfo->timeStamp;

        if (currentDelay < -lowerLimit)
            // Drop frame
            syncInfo->playDropHold    = tmolVrendGfxVoAvSyncDrop;
        else if (currentDelay > upperLimit)
            // Hold it
            syncInfo->playDropHold    = tmolVrendGfxVoAvSyncHold;
        else
            // Render it
            syncInfo->playDropHold    = tmolVrendGfxVoAvSyncPlay;
    }
}
```

```

    }

    return TM_OK;
}

```

A prerequisite is that `defaultSetup->clockHandle` is set to the MPEG clock when the video renderer is set up, as for the audio renderer.

The video renderer only calls the progress function when it is ready to render a frame (or field), which is at a frequency of 50Hz (NTSC), 60Hz (PAL), etc. The decision to play, drop or hold does not change this frequency, so we can see that coarse sync can never synchronize to a finer tolerance than this.

3.4 Video Decoder parameters

In order to improve performance, the video decoder also has the capability to drop frames. This gives an advantage in both processor cycles and memory usage and allows faster synchronization convergence. It is called Graceful Degradation.

The requirement of the video decoder for this are very similar to those of the video renderer. The differences are: (0) the Decoder never "holds" packets, (1) I-frames are never dropped, (2) the Decoder has to account for the fact that decoding itself takes a certain time. This time is called `DTS_SYNC_DELAY`. (It is modelled as 0 in the MPEG specification.)

The relevant part of the progress function is as follows:

```

static tmErrorCode_t progress
(
    Int          inst,
    tsaProgressFlags_t  flags,
    ptsaProgressArgs_t  args
)
{
    ptma1VdecMpeg_FrameInfo_t  frameInfo;
    Int          lowerLimit = ...,
                upperLimit = ...,
                currentDelay;
    tmTimeStamp_t  currTime;

    if ((args->progressCode & VDECMPPEG_PROGRESS_FRAME_INFO) != 0U)
    {
        frameInfo          = args->description;

        tsaClockGetTime (gPcrClock.clock, &currTime);
        currentDelay =
            frameInfo->Curr_Packet_Dts +
            DTS_SYNC_DELAY -
            currTime;

        if (currentDelay < -lowerLimit)
        {
            if (frameInfo->picType != I_FRAME)
                // Drop frame
                frameInfo->DtsDecodeCurrFrame =

```

```

        VDECMPEG_SKIP_CURRENT_FRAME;
    else
        // Never drop an I frame
        frameInfo->DtsDecodeCurrFrame =
            VDECMPEG_DECODE_CURRENT_FRAME;
    }
    else if (currentDelay > upperLimit)
        // Never hold
        frameInfo->DtsDecodeCurrFrame =
            VDECMPEG_DECODE_CURRENT_FRAME;
    else
        // Decode
        frameInfo->DtsDecodeCurrFrame =
            VDECMPEG_DECODE_CURRENT_FRAME;
    }
    return TM_OK;
}

```

A prerequisite is that `defaultSetup->clockHandle` is set to the MPEG clock when the video decoder is set up, as for the audio renderer.

Again, dropping frames leads to quite noticeable artefacts (jerky motion). And in general, all the techniques discussed so far are not acceptable during normal playback. They are only present in order to home in rapidly on the correct timing after a discontinuity, such as a channel change. According to the MPEG standard, there may be up to 1/3s delay introduced by the encoding process. So for full generality, the value of `SYNC_DELAY` should be ≥ 30000 or there is the risk of overruns or underruns.

3.5 Clock Recovery

The clock used by all the components we have described is the 90kHz MPEG clock. This has to match the PTSs of the streaming content in 2 ways. Firstly, the values of the time stamps in the data packets must correctly refer to the time at which the content should be presented. Secondly, the rate at which the clock ticks must match the rate at which the data is presented.

Note that this algorithm requires that the samples of the PCR given in the TS MUST match the arrival time noted by the FGPI hardware. If external hardware is used between the tuner and the FGPI, this timing may be affected hence invalidating the clock recovery algorithm.

Both of these requirements are met in the demux component, `tmDemuxMpegTs`. The component uses a "restamping" algorithm to make the values match. When it starts, it looks at the MPEG clock and at the PCRs in the transport stream. This difference is remembered and when a PTS is copied into an output packet, it is applied. The value of the local MPEG clock is never changed. This means that no other components (downstream) ever need to take account of clock discontinuities.

The same method also takes care of discontinuities in the PCRs of the transport stream itself. Every PCR in the stream is checked. If it lies in the valid range for a PCR at this point, it is used. Otherwise, it is assumed to be a discontinuity and the demux factors it out. (The allowed variation according to the MPEG specification is only 0.003%, but we can be more lenient than that.)

The remaining problem, frequency matching, can easily be understood as follows: what if the data is arriving faster (or slower) than it can be presented? This can happen if the clock at the encoder was running faster or slower than the true specification of 90kHz. The demux tracks this and modifies the frequency of the MPEG clock.

In order to force the MPEG clock to track the encoder clock (PCR) it needs a way of determining the real time at which each transport packet arrives. TriMedia input devices support 3 different ways of associating hardware time stamps with transport packets. Each has a different format subtype associated with it. (The class is `avdcSystem`, the type is `stfMPEG2Transport`) The demux looks at the subtype to work out where to get the hardware time stamp, as follows:

Subtype	Description
<code>tsfStandard</code>	regular MPEG2, sequence of 188-byte packets
<code>tsfTM2TimeStamped</code>	MPEG2 packet + 4-byte timestamp in cycles (obsolete)
<code>tsfFgpiTimeStamped</code>	4-byte timestamp in FGPI ticks + MPEG2 packet

These time stamps are used as follows: they recall that every PCR in the stream is checked. If it lies in the valid range for a PCR at this point, it is used (otherwise, it is assumed to be a discontinuity). If the PCR is higher than would be expected, the clock frequency is increased; if it is lower, decreased. (In the case of `tsfStandard`, there are no time stamps and no clock recovery.) The hardware time stamp is not seen by any component downstream of the demux. Its only effect is in the part it plays in frequency adjustment.

The demux also tracks the difference between the PCRs in the incoming stream and its PTSs. This difference is a measure of the sync delay at the encoder. It is called the "PCR correction". It is very jittery, as video frames can be out of order, but the demux filters out the jitter. It applies the PCR correction to the PTSs of packets it sends. This allows it to handle a greater variety of streams than it otherwise could at the expense of some extra coarse sync effects at the beginning of a new stream.

The effect of this is that the packet PTSs match the clock values when they arrive at the renderer. If we did nothing else, given that rendering a frame takes a fixed time, we would see a slow drift until the coarse sync algorithms defined above caused a packet to be dropped or held. This would result in that packets play locally at overall the same rate as at which they were encoded. For a high-quality system, relying on coarse sync for this level of control, is not acceptable, as it can lead to visible artifacts (a frame every few hours). In order to avoid this, we use fine sync.

If we are playing from file, the demux cannot do clock recovery. So the clock is left to run at 90kHz. In this case, there can still be a drift if the audio or video hardware is not presenting data at exactly the expected rate. It is possible to allow either the audio renderer or the video renderer to be the clock master in this case. Normally, the audio renderer is chosen. The result would be that the MPEG clock would track the audio PTSs and a frame of video might be dropped every few hours (or fine sync used) to keep in sync.

4 Fine sync

Fine sync is not yet implemented in `exolDemuxMpegTs`, so it is not described in detail here. See below for an overview.

It is conceptually very simple. If the renderer is running ahead of the packets, slow down the renderer; if behind, speed it up. The complication arises in knowing how much to speed up or slow down and when. Most of the problems can be solved very simply by a low-pass filter. (An example of such a filter can be found in the demux itself, where it is used to remove jitter in the PCR correction)

The extent to which this can be done is limited by the (narrow) bounds within which it is detectable by human perception. Luckily, the extent to which it is necessary is also narrow, because of the small tolerance allowed in the clock frequency.

For the video renderer the relevant call is the `tmolVrendGfxVoInstanceConfig()` command `VRENDGFXVO_CONFIG_CORRECT_FREQ`. This would be made in the progress function, if `syncInfo->playDropHold` was being set to `tmolVrendGfxVoAvSyncPlay` and it was determined that the clock was drifting.

The audio renderer also calls a progress function to allow the application to control the rendering speed. The progress flag is `ARENDSP_PROGRESS_SyncEventCorrect`. In this respect, it is much more like the video renderer. The application changes the ISR frequency by calling `tmolArendAOInstanceConfig()` with command `AR_SET_SAMPLE_RATE`. Again, there are limits to the allowed change. A quarter of a semitone is audible to many listeners and if the frequency has to be skewed by this much, it should take at least 5 seconds in total.

Conclusion

These techniques allow a very high-quality end result. The example application, `exolDemuxMpegTs`, demonstrates how they work together.