Book 5—System Utilities

Part D:

MPEG System Components

TriMedia

Version 2.0 beta

Table of Contents

Chapter 30 MPEG Transport Stream Demultiplexer (DemuxMpegTS) API

Chapter 32 VdigVIRaw API

Chapter 30

MPEG Transport Stream Demultiplexer (DemuxMpegTS) API

Note
This component library is not included with the basic TriMedia SDE, but is available as a part of other software packages, under a separate licensing agreement. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

DemuxMpegTS API Overview

The DemuxMpegTS module is an MPEG-2 demultiplexer module for MPEG-2 transport streams (see ISO/IEC 13818-1). The DemuxMpegTS component adheres to the standard TSSA component API.

The demultiplexer receives one MPEG-2 transport stream at a time. A transport stream consists of fixed-length packets of 188 bytes. Each packet has a header that contains a sync byte, an error indicator, and a packet identifier (PID) that identifies the stream to which the packet belongs, and other data. The demultiplexer scans the incoming transport stream for a sync byte and then starts decoding the stream. The demultiplexer can extract PIDs and send them to queues.

A transport stream can consist of multiple MPEG-2 programs, each described by a PMT (see the MPEG-2 standard). Each program usually consists of one or more audio PIDs and a video PID. The demultiplexer can extract more than one of these programs, and redirect the audio and video data to specified queues. The PIDs in certain programs refer to a system clock. The information necessary to re-generate this clock at the decoder's side is included in a PID that contains the Program Clock Reference (PCR). The demultiplexer extracts the timestamp information and maintains such a reference clock for each program. Audio and video decoders can inspect the re-generated clock and compare the timestamps of the decoded data in order to decide whether to present the decoded data. This comparison is required to obtain Audio Video synchronization (A/V sync).

The audio and video PIDs in a transport stream are MPEG-2 Packetized Elementary Streams (PES). The PES, an abstraction layer on top of the Elementary Stream (ES), is implemented with a header attached to an ES packet. The most important fields of this headers are the relation to the system clock. There can be two timestamps in such a PES header:

- The Presentation Timestamp (PTS) that specifies the time at which the data are to be presented.
- The Decoding Timestamp (DTS) that specifies the time at which the data are decoded.

The demultiplexer conforms to the standard TSSA interface: it has the standard error and progress reporting and its configuration can be modified through the control queue.

Limitations

The demultiplexer parses large amounts of data. Typically, the data rates are on the order of a few megabytes per second. To do its job efficiently, the demultiplexer does not copy the data. Instead, it passes pointers to buffers that hold the incoming bitstream as TSSA 'full' packets to the audio and video components. The demultiplexer must see all these packets returned (as TSSA 'empty' packets) before it can tag the input packet as empty. Once an input packet is tagged as empty, it will be returned to the empty input queue. When one of the audio or video components holds on to the full packets too long, the

demultiplexer pipeline stops, simply because the demultiplexer does not return empty packets to its input queue anymore. The requirement that the downstream components send the packets back as soon as possible is a limitation that holds only for the audio and video outputs. The other outputs get copies of the data. This should not be a problem since the data rates are much lower.

Note 1
Downstream audio and video decoders must send empty packets back to the demultiplexer as soon as possible.

Note 2
Although the demultiplexer has an application layer (see TSSA) it is primarily accessed through the tmol layer. The tmal layer is therefore not documented.

Features

- The demultiplexer prefetches data into the cache before sending it to the audio and video decoders.
- The DemuxMpegTS component is re-entrant. Multiple instances can be running at the same time.
- The demultiplexer can dynamically create additional outputs (see RC-5 Inputs and Outputs). This means that handling certain private data streams in a later development stage is possible, without changes to the demultiplexer.
- The demultiplexer has plug-ins for determining whether data must be sent to a queue that is not an audio/video queue. These plug-ins can help implement section filters.
- The demultiplexer can extract up to 4 programs (each containing video, audio, secondary audio and clock reference).
- The demultiplexer manages its own memory.
- The demultiplexer input can either be connected to a generic file reader or DMA-like device or handle the TM-2*xxx* transport block output.

Clock Recovery

The demultiplexer maintains clocks for all the selected PCR PIDs that are requested. The clock maintained internally is a 27 MHz clock, and the clock that is exposed through a clockHandle is a 90 kHz clock derived from that 27 MHz clock. The PCRs are extracted from the bitstream and then converted to the TriMedia clock. Using this mechanism, the decoder clock at the TriMedia processor is maintained at the same frequency as the reference clock in the encoder.

Although the frequency is the same as the encoder clock, the value is not. Rather than adjusting the value of the clock, offsets are added to the PTSs sent to the audio and video packets. This means that the regenerated clock cannot be directly inspected for timestamps other than the ones that are extracted by the demultiplexer (such as the audio

and video PTSs and DTSs). There are two offsets maintained by the demultiplexer. The first is the one that is the offset to the original PCR, to adjust for the value not being like the value of the eoncoder's clock. The second is an offset that is added to the PTSs to allow the software audio or video decoders to do their work. Basically, the demultiplexer tries to keep the time between PCR and PTS in a certain range over all bitstreams. With the instance setup variables, an application can change this offset, and so tune the number of buffers needed in the system. This last offset is not modified frequently (for correct execution, it should be modified only at bitstream change or startup) since changing this offset is clearly audible and visible.

The demultiplexer sends out the offsets for packets that are not audio and video packets. The timestamps in those packets contain the two offsets. Subtracting the offset in the **time.ticks** from the actual regenerated clock value from the clockHandle will result in the absolute value of the encoders' clock. (See *Outputs on page* 80.)

DemuxMpegTS Inputs and Outputs

Overview

Figure 5 shows the input and outputs of the demultiplexer. The one data input is the MPEG-2 transport stream. The AV outputs are grouped in sets of 3 queues, one for video, one for main audio and one for secondary audio. Each of these might or might not be active. A control input can be used to redirect audio/video elementary streams, PSI data or other private data to certain queues. Other outputs are the error report and progress reports in which the demultiplexer reports some status. All outputs are TSSA bidirectional queues.

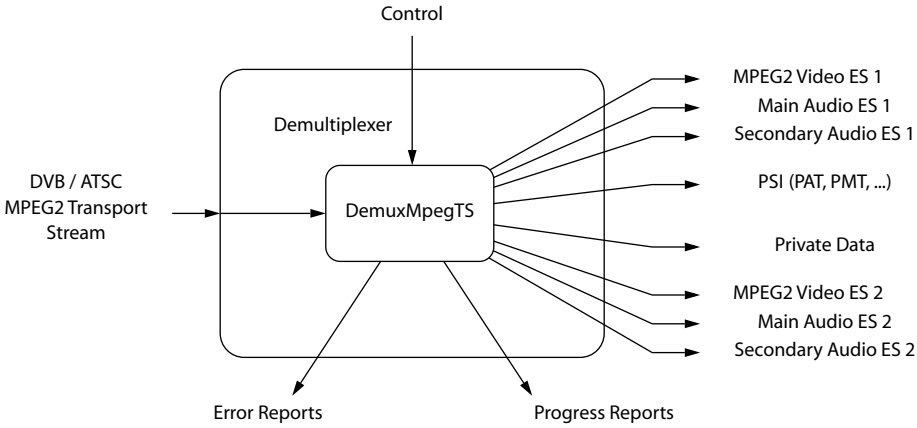

Figure 1 Overview of the Demultiplexer

Inputs

The demultiplexer takes one input. That is the MPEG-2 transport stream. This stream is normally retrieved from the Video-In peripheral or the Transport Stream Input block for the TM-2*xxx* family, but can be read over PCI or coming in over 1394 also. The demultiplexer expects input in large buffers. It makes no assumptions about the position of the sync byte (unless coming from the transport block where sync byte position is guaranteed) or about buffer sizes.

The incoming packets are of type **tmAvPackets_t**. From the format of the packet, it is determined what fields are expected to be set. The capability format for the input descriptor is set to

```
tmAvFormat_t tpInFormat = {
   sizeof(tmAvFormat_t),                                  /* size           */
   0,                                                     /* hash           */
   0,                                                     /* referenceCount */
   avdcSystem,                                            /* dataClass      */
   stfMPEG2Transport,                                     /* dataType       */
   tsfStandard | tsfTM2TimeStamped | tsfStandard204,      /* dataSubtype    */
   0                                                      /* description    */
};
```

For packets with the **dataSubtype** field set to **tsfStandard**, the data stream is expected to be a digitized MPEG-2 transport stream, packets of 188 bytes with appropriate sync bytes, as described in ISO/IEC 1318-1. The following fields are expected to be set:

`time.hiTicks`	The start timestamp which is the time stamp of the TriMedia clock (**cycles** custom_op) at the start of buffer capture (or the end of the previous buffer capture).
`time.ticks`	contains the end timestamp which is the time stamp of the TriMedia CPU clock (**cycles** custom_op) at the end of the buffer capture.

These fields must be set in order to do clock recovery. (For file-based inputs over PCI, this information cannot be accurate and therefore clock recovery will not be accurate.)

For packets with the **dataSubtype** field set to **tsfTM2TimeStamped**, the data stream is expected to be a digitized MPEG-2 transport stream with packets of 192 bytes. The last four bytes contain an error bit and a timestamp as described in the TM-2*xxx* Transport Stream Block description.

For packets with the **dataSubtype** field set to **tsfStandard204**, the data stream is expected to be a digitized MPEG-2 transport stream with packets of 204 bytes, like they are being used in the Japanese BS digital standard.

Note
For correct clock recovery for Video-In or PCI-based transport stream inputs, the time structure in the tmAvHeader of the tmAvPacket must be set to appropriate time stamps.

The input descriptor has index 0. See **tmalDefaultCapabilities_t**.

```
#define DEMUXMPEGTS_INPUT 0
```

Outputs

The demultiplexer acts on requests from the application environment to produce outputs. There can be requests for AV outputs and request for other PIDs that have, for instance, PSI information (PAT and PMT). The AV PIDs are requested through a set of interface routines that has an index as parameter that is taken as an offset. The offset is relative to one of the following base outputDescriptor indices:

```
DEMUXMPEGTS_VIDEO_OUTPUT
DEMUXMPEGTS_AUDIO_OUTPUT
DEMUXMPEGTS_SECONDARY_AUDIO_OUTPUT
```

You may view the index as a program identifier, and request a group of one video, one PCR, one audio and one secondary audio PIDs to be redirected to index 1, which, for instance, would redirect the video PID to the outputDescriptor at index **DEMUXMPEGTS_VIDEO_OUTPUT1.**

You can pass a queue together with a PID. The demultiplexer will redirect copies[1] of the data of that PID to that queue. You can request any PID to be redirected, but each PID, including the audio and video PIDs, can be redirected to one queue only. You can request elementary stream packets for audio and video PIDs, raw transport packets, or you can instruct the demux to assemble MPEG-2 sections. For redirected sections, you can request the demultiplexer to perform the CRC. You must insert empty packets with pre-allocated buffers in the queue. The pre-allocated buffers must be large enough for the unit (transport packets of 188 bytes or the maximum section size) that is to be redirected to this queue, because the demultiplexer copies the data into the packet.

For redirected MPEG-2 sections, a callback function can be supplied as a filter that tells the demultiplexer whether or not to pass the data to the queue. This is useful for DVB section filtering. The demultiplexer assembles the section (with optionally a CRC), checks through the callback whether this section needs to be passed along, and then sends or discards the section.

As mentioned previously, non-audio and non-video packets have a time stamp that is not a real timestamp but rather two offsets. These two offsets are used within the demultiplexer to offset the PTSs from the PCR clock. (See *Non-AV outputs* on page 14.)

For details of how to request extra outputs, see **tmolDemuxMpegTSAddRedirectedPid**.

Every output you do not want can be left unused. When an output is not needed, such as private data, you can simply not create the IODescriptor and not overwrite the output in the array of outputDescriptors. However, all audio and video outputs and other queues that will be used must be fully initialized during instance setup. The demulti-

1. Note that the audio and video outputs of the demultiplexer contain pointers to data and not actual copies. You can request audio and video data explicitly by the redirection method, but you must consider that this will take a large number of cycles when the HD video stream is copied.

plexer requires all audio and video empty packets to be resident in the queue at the call to **tmolDemuxMpegTSStart**. It is not possible to install IODescriptors at run-time and then call instance setup again. It is also not possible to create more audio and video empty packets on-the-fly.

Before the demultiplexer sends data pointers along (for the audio and video outputs of the demultiplexer) it prefetches the data. Components can assume the data is in the cache on arrival of the packets.

The output descriptor assignment is:

```
#define DEMUXMPEGTS_VIDEO_OUTPUT             0
#define DEMUXMPEGTS_VIDEO_OUTPUT0            0
#define DEMUXMPEGTS_VIDEO_OUTPUT1            1
#define DEMUXMPEGTS_VIDEO_OUTPUT2            2
#define DEMUXMPEGTS_VIDEO_OUTPUT3            3
#define DEMUXMPEGTS_AUDIO_OUTPUT             4
 and similar to video OUTPUT0...OUTPUT3
#define DEMUXMPEGTS_SECONDARY_AUDIO_OUTPUT   8
 and similar to video OUTPUT0...OUTPUT3
#define DEMUXMPEGTS_PSI_OUTPUT              12
#define DEMUXMPEGTS_PRIVATE_DATA_OUTPUT     13
```

Note
The demultiplexer requires all queues and clockHandles that are used to be initialized (non-null and queues with empty packets inserted) at instance setup.

Audio and Video Outputs

The audio and video outputs of the demultiplexer have the default **tmAvPacket_t** structure with the following field set in the header (**tmAvHeader_t**):

`flags` — The **avhValidTimestamp** bit is set when a valid presentation timestamp (PTS) is attached to this packet. The **avhValidDts** bit is set when the timestamp of the header contains a valid decoding timestamp (DTS). This packet will have no data (number of buffers set to 0) and the DTS applies to the next packet with a valid PTS (**avhValidTimestamp** bit set).

`time.ticks` — The lower 32 bits of the PTS timestamp as stored in the PES header of the audio/video PID. (See the MPEG-2 standard.)

`time.hiTicks` — The 33rd bit of the PTS timestamp as stored in the PES header of the audio/video PID. Currently the software sets this bit to 0; only 32 bits of the timestamp are used.

The demultiplexer does not analyze the PMT. It does not know to what elementary stream it will send, for instance, the audio queue. Since TSSA requires formats to be installed on the IODescriptors, it is the responsibility of the application to install the correct format on the queue. In the case of audio, this might be, for instance, AC3 data or MPEG-1 level 2 audio.

The output capability format for the audio output is:

```
tmAvFormat_t audioFormat = {
   sizeof(tmAvFormat_t),  /* size           */
   0,                     /* hash           */
   0,                     /* referenceCount */
   avdcAudio,             /* dataClass      */
   atfAC3 | atfMPEG,      /* dataType       */
   amfGeneric,            /* dataSubtype    */
   0                      /* description    */
};
```

The output capability format for the video output is:

```
tmAvFormat_t videoFormat = {
   sizeof(tmAvFormat_t),  /* size           */
   0,                     /* hash           */
   0,                     /* referenceCount */
   avdcVideo,             /* dataClass      */
   vtfMPEG,               /* dataType       */
   vmfMPEG2,              /* dataSubtype    */
   0                      /* description    */
};
```

Non-AV outputs

Dynamically redirected outputs can be of two types: MPEG-2 sections and raw transport packets. The dynamically redirected outputs with section type are similar to PSI sections sent to the PSI output of the demultiplexer, whereas the redirected outputs of transport packet type are similar to private data from the adaptation field sent to the private data output of the demultiplexer.

The non-AV outputs copy data into pre-allocated buffers. Your buffers must be big enough to hold the maximum length output. For demuxMpegTSTransport packets, the maximum length is 188 bytes, but for the PSI sections (PAT and PMT and other sections) the maximum length is usually 1024 bytes. When PSIP (ATSC) sections are requested, these buffers must be 4096 bytes. You must supply a sufficient number of packets in the empty queue and make sure that the scheduling of tasks allows non AV-packets to be processed in a timely manner. The demultiplexer will block on an empty non AV-empty-queue but will timeout after 1 OS clock tick. Data will be lost if your application does not prevent this from happening.

The non-AV outputs of the demultiplexer have the default **tmAvPacket_t** structure with the following field set in the header (**tmAvHeader_t**):

`userSender` — The PID number for which this section is retrieved. Together with the **table_id** from the section, this should determine the type of the packet. If you are using a callback function, the returned **userDataOutput** value is put in the **userSender** field.

`time.hiTicks`	An offset that the demultiplexer adds to the PTSs to correct for the software audio and video decoders. See the instance setup variables **decodersPcrShiftLow** and **decodersPcrShiftHigh**.
`time.ticks`	An offset that, when subtracted from the regenerated clock value, would result in the actual PCR value.

More information can be found under *Clock Recovery* on page 9.

The output format for these outputs are:

```
tmAvFormat_t demuxDataFormat = {
   sizeof(tmAvFormat_t),  /* size           */
   0,                     /* hash           */
   0,                     /* referenceCount */
   avdcGeneric,           /* dataClass      */
   avdtGeneric,           /* dataType       */
   avdsGeneric,           /* dataSubtype    */
   0                      /* description    */
};
```

DemuxMpegTS Errors

The demultiplexer handles all bitstream errors. It reports and handles transmission errors and MPEG-2 standard violations. Mostly, these errors cause skipping of data. It is expected that downstream components handle erroneous data as well.

System errors such as memory allocation or OS errors are reported as system failures and the system must take appropriate action.

The description on page 93 has more information regarding error codes.

The demultiplexer error function is a standard TSSA error function, and is installed as a callback function during instance setup. The type of the function is given below. The field **args->errorCode** can be cast to the type **tmalDemuxMpegTSErrorFlags_t**, and the **args–>description** value, when set and not explicitly explained, can be interpreted as line number.

```
tmLibappErr_t
DemuxMpegTSError( Int instId, UInt32 flags, ptsaErrorArgs_t args )
```

Note
Downstream audio and video decoders must handle bitstream errors as well as the demultiplexer and must not stall the pipeline.

DemuxMpegTS Progress

The demultiplexer has 5 progress reports.

1. A request to change the audio-out and video-out clock frequencies. The demultiplexer regenerates the 27 MHz clock of the encoder through the PCR (Program Clock Reference). It can happen that, when the TriMedia processor's clock and the encoder's clock do not exactly match in frequency, the demultiplexer requests to change the audio and video clocks in order not to run ahead or behind. The two clocks that must be adjusted are the Audio-Out peripheral clock, through the audio renderer, and the Video-Out clock. Alternatively, the audio and video decoders can lock their clocks directly to the PCR, in which case this progress report need not be used.
2. A progress report to indicate that a discontinuity signalled on one of the non-AV PIDs. This indicates a discontinuity as stated in the MPEG-2 standard section 2.4.3.5.
3. A progress report for timeouts that occurred on the input queue, that is, an underrun has occurred on the TSSA full input queue. This condition is reported only when the demultiplexer does not have all the input buffers in its possession. The input stalled for an unknown reason. The system can take appropriate action.
4. A lost sync progress report. The demultiplexer expects sync at certain positions in the bitstream and when it cannot find one, it reports the fact. This error can occur because of a data error or because of other failures in the system. The demultiplexer then starts looking for a new sync byte itself.
5. A progress report that reports the change in offset that is added to the PTS values. This offset is determined via decodersPcrShiftLow and decodersPcrShiftHigh, and ideally should be set by the demultiplexer only once for each bitstream. With these offsets the number of buffers needed in the system can be tuned. But sometimes adjustments may be necessary and these adjustments are probably visible and audible. This progress report indicates such an action taken by the demux and also reports the new offset to the application.

More information can be found under **tmalDemuxMpegTSProgressFlags_t** on page 98. The progress report function is a standard TSSA callback function installed during instance setup. The type of the function is given below. The **args–>progressCode** argument can be cast to the type **tmalDemuxMpegTSProgressFlags_t** and the **args–>description** value is described in the **tmalDemuxMpegTSProgressFlags_t** data structure.

```
tmLibappErr_t
DemuxMpegTSProgress( Int instId, UInt32 flags, ptsaProgressArgs_t args )
```


DemuxMpegTS Configuration

The demultiplexer handles the following requests through a control queue:

1. The default TSSA function status.
2. Change the PCR PID for a certain index (as in *index* of the group of outputDescriptors, as discussed under *Outputs* on page 80).
3. Change the video, audio, or secondary audio PID for a certain index (0 ≤ index < **DEMUXMPEGTS_NROF_AV_OUTPUTS**).
4. Add or delete redirection of a certain PID. (See *RC-5 Inputs and Outputs*.)

At the OL layer, this control is provided by a functional interface that communicates with the AL layer through a synchronous control queue. More information on the control functions can be found in the description of the following functions:

```
tmolDemuxMpegTSChangeVideoPid,
tmolDemuxMpegTSChangePcrPid,
tmolDemuxMpegTSChangeMainAudioPid,
tmolDemuxMpegTSChangeSecondaryAudioPid,
tmolDemuxMpegTSChangeToNewPids
tmolDemuxMpegTSAddRedirectedPid,
tmolDemuxMpegTSRemoveRedirectedPid.
```

More information on which parameters must be passed to the queue can be found under **tmalDemuxMpegTSControlArgs_t** on page 30.

Note
The control functions are synchronous. Do not call them from the installed error function. Calling them from the progress function or from within the section filter is permissible.

DemuxMpegTS API Data Structures

This section presents the DemuxMpegTS component data structures.

tmalDemuxMpegTSInstanceSetup_t

```
typedef struct tmalDemuxMpegTSInstance {
   ptsaDefaultInstanceSetup_t   defaultSetup;
   ptmalDemuxMpegTSConfig_t     demuxConfig;
} tmalDemuxMpegTSInstanceSetup_t, *ptmalDemuxMpegTSInstanceSetup_t;
```

tmolDemuxMpegTSInstanceSetup_t

```
typedef tmalDemuxMpegTSInstance    tmolDemuxMpegTSInstance;
typedef ptmalDemuxMpegTSInstance   ptmolDemuxMpegTSInstance;
```

Fields

`defaultSetup`	See TSSA documentation.
`demuxConfig`	See **tmalDemuxMpegTSConfig_t** on page 90.

Description

The data structure passed to **tmolDemuxMpegTSInstanceSetup** or **tmalDemuxMpegTS-InstanceSetup** to describe the input and output connections and other initial values.

tmalDemuxMpegTSCapabilities_t

```
typedef struct tmalDemuxMpegTSCapabilities{
   ptsaDefaultCapabilities_t   defaultCaps;
} tmalDemuxMpegTSCapabilities_t, *ptmalDemuxMpegTSCapabilities_t;
```

Fields

`defaultCaps`	See TSSA documentation.

Description

For input and output descriptors, see *RC-5 Inputs and Outputs* on page 60. The text section of the demultiplexer is about 50 kb and the data required for a single instance is about 30 kb.

tmalDemuxMpegTSConfig_t

```
typedef struct tmalDemuxMpegTSConfig {
   UInt32              nrofInputBuffers;
   UInt32              inputBufferSize;
   UInt32              ticksPerSecond;
   Bool                ignoreSoftError;
   Float               incomingDataRate;
   UInt32              decodersPtsShiftLow;
   UInt32              decodersPtsShiftHigh;
   ptsaClockHandle_t   clockHandles[DEMUXMPEGTS_NROF_AV_OUTPUTS];
} tmalDemuxMpegTSConfig_t, *ptmalDemuxMpegTSConfig_t;
```

Fields

`nrofInputBuffers` — Number of input buffers between the input producer (e.g., video-in or TM-2*xxx* transport block) and the demultiplexer.

`inputBufferSize` — Size of the input buffer (must be the same for all buffers).

`ticksPerSecond` — Number of ticks per second the OS clocks provides (for pSOS, this is **KC_TICKS2SEC**). Together with **inputBufferSize**, this value is used to calculate a datain timeout.

`ignoreSoftError` — Ignores soft errors (**MPEG-2** standard violations), such as reserved bits not matching the specified values, etc. See **tmalDemuxMpegTSErrorFlags_t**.

`incomingDataRate` — A value for calculating when to send the **DEMUXMPEGTS_NO_INCOMING_DATA** progress report. Cannot be changed on-the-fly.

`decodersPtsShiftLow`, `decodersPtsShiftHigh` — The lower and upper bounds of the difference, between the PCR and the PTSs, the demultiplexer must keep. This shift is required to correct for software decoder delays in the system and to keep the number of buffers of audio/video elementary stream data nearly constant over a diversity of input streams. (The value is specified in number of ticks of the MPEG-2 90 kHz clock.)

`clockHandles` — Array of clock handles. At least **clockHandles[0]** must be set to a valid TSA clock handle as normally would be used in the defaultSetup's **clockHandle**. Since the demultiplexer uses more than one clock handle, it does not use the clock handle from the **tsaDefaultInstanceSetup_t**. A valid clock handle must be set for each index used (that is, a

valid **pcrPid** is set) during the lifetime of this demultiplexer's instance. Note that the regenerated clock is a 90 kHz clock, but the value of this clock may not be close to the value of the actual PCR values. The PTSs passed to the audio and video channels simply have offsets from the regenerated clock and can also not be close to the actual PTS values.

Description

Controls the demultiplexer's task-level instance setup and is used by the **tmolDemuxMpegTSInstanceSetup** function. These fields cannot be changed after instance setup, because instance setup for the demultiplexer can be called only once for each instance.

tmalDemuxMpegTSErrorFlags_t

The abbreviation **Err_base** is used here to stand for **Err_base_DEMUXMPEGTS**, which is 0x11050000.

```
typedef enum {                                  /* Fatal errors */
  DEMUXMPEGTS_ERR_INVALID_NROF_BUFFERS         = Err_base + 0x0001,
  DEMUXMPEGTS_ERR_INVALID_BUFFER_SIZE          = Err_base + 0x0002,
  DEMUXMPEGTS_ERR_INVALID_OS_CLOCK_TICK_VALUE  = Err_base + 0x0003,
  DEMUXMPEGTS_ERR_INVALID_CLOCK_HANDLE         = Err_base + 0x0004,
  DEMUXMPEGTS_ERR_NONE_SECTION_LENGTH          = Err_base + 0x0005,
  DEMUXMPEGTS_ERR_BUFFERS_DATA_FIELD_NON_ZERO  = Err_base + 0x0101,
  DEMUXMPEGTS_ERR_DATA_FIELD_ZERO              = Err_base + 0x0102,
  DEMUXMPEGTS_ERR_INVALID_QUEUE_INDEX          = Err_base + 0x0103,
  DEMUXMPEGTS_ERR_INVALID_BUFFERS_IN_USE       = Err_base + 0x0104,
  DEMUXMPEGTS_ERR_TM2_INPUT_EXPECTED           = Err_base + 0x0105,
  DEMUXMPEGTS_ERR_NO_OUTPUTDESCRIPTOR          = Err_base + 0x0106,
  DEMUXMPEGTS_ERR_PCR_SHIFT_RANGE_TOO_SMALL    = Err_base + 0x0107,
  DEMUXMPEGTS_ERR_INTERNAL_ERROR               = Err_base + 0x011F,
/*Non-fatal errors. Demultiplexer just reports them.
 *Data can be lost at the point of error. */
  DEMUXMPEGTS_ERR_PID_NOT_FOUND                = Err_base + 0x0201,
  DEMUXMPEGTS_ERR_INVALID_COMMAND              = Err_base + 0x0202,
  DEMUXMPEGTS_ERR_NO_EMPTY_PACKET              = Err_base + 0x0203,
  DEMUXMPEGTS_ERR_INVALID_REQUESTED_PID        = Err_base + 0x0204,
  DEMUXMPEGTS_PES_HEADER_LENGTH_TOO_LONG       = Err_base + 0x0205,
  DEMUXMPEGTS_PES_DATA_LENGTH_TOO_LONG         = Err_base + 0x0206,
  DEMUXMPEGTS_PES_INVALID_STARTCODE            = Err_base + 0x0207,
  DEMUXMPEGTS_PES_TIME_STAMP_MARKER_BITS       = Err_base + 0x0208,
  DEMUXMPEGTS_CONTINUITY_COUNTER_MISMATCH      = Err_base + 0x0209,
  DEMUXMPEGTS_ERROR_IN_PACKET                  = Err_base + 0x0210,
  DEMUXMPEGTS_ADAPTATION_FIELD_LENGTH_TOO_LONG = Err_base + 0x0211,
  DEMUXMPEGTS_ADAPTATION_FIELD_LENGTH_MISMATCH = Err_base + 0x0212,
  DEMUXMPEGTS_PRIVATE_DATA_LENGTH_TOO_HIGH     = Err_base + 0x0213,
  DEMUXMPEGTS_SECTION_SIZE_TOO_LONG            = Err_base + 0x0214,
  DEMUXMPEGTS_TIME_STAMP_MARKER_BITS           = Err_base + 0x0215,
/*Non-fatal Soft Errors. Can be ignored with flag  passed to
 *instance setup. See ignoreSoftError */
  DEMUXMPEGTS_DEMUX_ADAPTATION_RESERVED_BITS   = Err_base + 0x0400,
} tmalDemuxMpegTSErrorFlags_t;
```

Fields

(Fatal errors by the demultiplexer)

`DEMUXMPEGTS_ERR_INVALID_NROF_BUFFERS`

During instance setup, **nrofInputBuffers** ≤ 0. Triggered as assert.

`DEMUXMPEGTS_ERR_INVALID_BUFFER_SIZE`

During instance setup, **inputBufferSize** ≤ 0, or the value is so small that the demultiplexer cannot find a number of correctly spaced sync bytes in one buffer. Typically, 3 sync bytes must be 188 bytes apart in order to determine synchronization. These are expected to fall within one buffer. Triggered as assert.

`DEMUXMPEGTS_ERR_INVALID_CLOCK_TICK_VALUE`

During instance setup, the demultiplexer is passed a value ≤ 0. Triggered as assert.

`DEMUXMPEGTS_ERR_INVALID_CLOCK_HANDLE`

(1) During instance setup, the demultiplexer was not passed a valid clock handle in the **demuxConfig.clockHandle[0]**, (2) the **defaultSetup.clockHandle** was set to a value other than **Null**, or (3) a **ChangePcrPid** was requested with an index for which no clockHandle was provided at instance setup. Triggered as assert for instance setup, and returned as a function value for **ChangePcrPid**.

`DEMUXMPEGTS_ERR_NONE_SECTION_LENGTH`

A buffer was passed for a redirected PID of type **demuxMpegTSTransport**. The packets in the queue have an allocated data buffer that is too small to copy the data. Packets must have buffers of at least **DEMUXMPEGTS_MAX_NONE_SECTION_LENGTH** (188) bytes.

`DEMUXMPEGTS_ERR_BUFFERS_NON_ZERO`

During initialization, you probably passed data pointers for the A/V Packets. The demultiplexer sends pointers, and expects the data pointers to be null. Triggered as assert.

`DEMUXMPEGTS_ERR_DATA_FIELD_ZERO` There is no memory allocated for the PSI or private data or extra requested PID packets. When putting packets in non-audio/video queues, you must allocate memory and supply sufficient packets. Triggered as assert.

`DEMUXMPEGTS_ERR_INVALID_QUEUE_INDEX`

(1) A queue index exceeded the maximum number of output queues of the demultiplexer for **AddRedirectedPid**, or (2) an invalid index was used in one of the **Change*xxx*Pid** functions. Nonfatal return value.

`DEMUXMPEGTS_ERR_INVALID_BUFFERS_IN_USE`
: The demultiplexer does not handle more than one buffer per packet in its input queue. Triggered as assert.

`DEMUXMPEGTS_ERR_TM2_INPUT_EXPECTED`
: Although the format of an input packet indicated a TM-2*xxx* Transport Stream Block input type (the **subType** of the format was **tsfTM2TimeStamped**), the timestamp is placed unaligned, so probably the format is wrong.

`DEMUXMPEGTS_NO_OUTPUTDESCRIPTOR` A PID was redirected to a queue for which there was no output descriptor installed.

`DEMUXMPEGTS_ERR_PCR_SHIFT_RANGE_TOO_SMALL`
: The gap between **decodersPtsShiftHigh** and **decodersPtsShiftLow** should be at least 3000 ticks. There must be a gap big enough to allow a variation in the bitstream without constantly updating the software decoder delay offset, since modifying this offset is audible and visible.

`DEMUXMPEGTS_ERR_INTERNAL_ERROR` An internal error of unspecified origin occurred. Contact the vendor. Triggered as assert.

The following are non-fatal errors reported by the demultiplexer. Some of these occur because of errors in the bitstream. Some data may be lost because of the error.

`DEMUXMPEGTS_ERR_PID_NOT_FOUND` A PID that is requested for redirection could not be found during a call to **RemoveRedirectedPid**.

`DEMUXMPEGTS_ERR_INVALID_COMMAND` An unknown command is passed through the control queue. Non-fatal return value.

`DEMUXMPEGTS_ERR_NO_EMPTY_PACKET` The demultiplexer could not get an empty packet for the PSI or requested outputs. Data will be lost. The third argument to the error report function is the queue index for which the error occurred.

`DEMUXMPEGTS_ERR_INVALID_REQUESTED_PID`
: (1) A PID is already allocated to another queue when adding a user-requested redirection of a PID, or (2) the PID can not be found when deleting a user-requested PID. Non-fatal return value.

`DEMUXMPEGTS_PES_HEADER_LENGTH_TOO_LONG`
: PES header length is longer than the maximum allowed length. The packet is discarded and PES header parsing is re-initialized.

`DEMUXMPEGTS_PES_DATA_LENGTH_TOO_LONG`
: Data too long for the **PRIVATE_STREAM_2** or **PADDING_STREAM** stream types. The packet discarded.

`DEMUXMPEGTS_PES_TIME_STAMP_MARKER_BITS`
: Invalid marker bits in the PTS or DTS. Timestamp made invalid.

`DEMUXMPEGTS_CONTINUITY_COUNTER_MISMATCH`
: A continuity counter error occurred. (These are not the duplicate packets.)

`DEMUXMPEGTS_ERROR_IN_PACKET`
: The error bit in the packet was set. The packet is discarded.

`DEMUXMPEGTS_ADAPTATION_FIELD_LENGTH_TOO_LONG`
: Adaptation field length is longer than 187 bytes.

`DEMUXMPEGTS_ADAPTATION_FIELD_LENGTH_MISMATCH`
: The bytes read by the adaptation field parser do not match the specified length. Adaptation field is discarded.

`DEMUXMPEGTS_ADAPTATION_RESERVED_BITS`
: The reserved bits values do not match the values specified by the standard.

`DEMUXMPEGTS_PRIVATE_DATA_LENGTH_TOO_HIGH`
: Private data longer than 184 bytes.

`DEMUXMPEGTS_SECTION_SIZE_TOO_LONG`
: A section-specified length exceeds 1024 bytes. The TSSA standard allows 1021 maximum.

`DEMUXMPEGTS_TIME_STAMP_MARKER_BITS`
: Marker bits in a timestamp are invalid (timestamp for the splice points).

Description

The demultiplexer expects the application to handle system errors. Non-fatal errors can be ignored or you can use them to re-tune the incoming frequency or start error concealment in the video decoder. The demultiplexer handles the errors internally, such that it continues to parse the transport stream the best way possible. Decoder and other components that get data from the demultiplexer must also be able to handle erroneous data.

tmalDemuxMpegTSProgressDescription_t

```
typedef struct tmalDemuxMpegTSProgressDescription {
   UInt32 size;
   union {
      struct {
         UInt32 index;
         Float  new27MHzFrequency;
      } newClockFrequency;
      UInt32 pid;
      struct {
        UInt32 index;
        UInt32 offset;
      } newOffset;
   } args;
} tmalDemuxMpegTSProgressDescription_t,
*ptmalDemuxMpegTSProgressDescription_t;
```

Fields

`size`	The size of the structure as required by tsa.
`newClockFrequency`	Structure used when progress code is **DEMUXMPEGTS_NEW_CLOCK_FREQUENCY**.
`index`	Index to which the progress report applies. (0 ≤ **index** < **DEMUXMPEGTS_NROF_AV_OUTPUTS**)
`new27MHzFrequency`	The new 27 MHz clock frequency.
`pid`	Used when the progress code is **DEMUXMPEGTS_DISCONTINUITY**.
`newOffset`	Structure used when progress code is **DEMUXMPEGTS_NEW_PTS_OFFSET**.
`index`	Index to which the progress report applies. (0 ≤ **index** < **DEMUXMPEGTS_NROF_AV_OUTPUTS**)
`offset`	The new offset, which is added to each PTS sent to audio and video decoders and copied to **time.hiTicks** of other packets sent out. See *Non-AV outputs* on page 14 and *Clock Recovery* on page 9.

Description

This structure is passed as the description field of the **tsaProgressArgs_t**.

tmalDemuxMpegTSProgressFlags_t

```
typedef enum {
   DEMUXMPEGTS_NEW_CLOCK_FREQUENCY = 0x00000001,
   DEMUXMPEGTS_DISCONTINUITY       = 0x00000002,
   DEMUXMPEGTS_NO_INCOMING_DATA    = 0x00000004,
   DEMUXMPEGTS_LOST_SYNC           = 0x00000008
} tmalDemuxMpegTSProgressFlags_t;
```

Fields

`DEMUXMPEGTS_NEW_CLOCK_FREQUENCY` The demultiplexer's clock recovery module has found that the frequency of the audio and video decoders needs to be adjusted to operate correctly.

In **args.description**, use the **newClockFrequency** field of **tmalDemuxMpegTSProgressDescription_t**.

Action: The controlling application must adjust the frequencies of the audio and video decoder clocks appropriately. This usually means that the audio frequency in audio out is set to 48 kHz × args / 27 MHz and the video decoder clock is sped up or slowed down similarly. Some applications might ignore this report and adjust the output clocks based on the PCR and PTS differences.

`DEMUXMPEGTS_DISCONTINUITY` A discontinuity is signalled in the adaptation field of a redirected PID. This PID is not an audio or video PID and is possibly a PMT PID signalling a version number discontinuity according to the MPEG-2 standard section 2.4.3.5.

In **args.description**, use **tmalDemuxMpegTSProgressDescription_t** to get the PID.

Action: None is expected by the demultiplexer.

`DEMUXMPEGTS_NO_INCOMING_DATA`

A timeout on the input queue occurred. The timeout is set to

$$1 + \frac{\text{ticksPerSecond}}{\text{incomingDataRate / inputBufferSize}}$$

All of these variables are instance variables and can be set by the user. The demultiplexer only reports this error when it does not have all the input buffers, so there is something wrong at the input.

Action: inspect the system for errors. Either the program has ended or we have lost data.

DEMUXMPEGTS_LOST_SYNC	The demultiplexer must start looking for a sync byte because it did not occur at the expected location in the data. *Action:* ignore this or start error concealment.

tmalDemuxMpegTSRedirectedOutputFormat_t

```
typedef enum {
   demuxMpegTSTransport,
   demuxMpegTSSection,
   demuxMpegTSSectionCRC
} tmalDemuxMpegTSRedirectedOutputFormat_t;
```

Fields

demuxMpegTSTransport	Request MPEG-2 transport packets.
demuxMpegTSSection	Request MPEG-2 sections, no CRC. PID is stored in **userSender** field.
demuxMpegTSSectionCRC	Request MPEG-2 sections, with CRC. When CRC does not match, section is discarded. PID is stored in **userSender** field.

Description

Enumerates the type of data that will be put in the dynamically redirected outputs of the demultiplexer. Extra redirected outputs can be requested by calling **tmolDemuxMpegTS-AddRequestedPid**.

tmalDemuxMpegTSSectionCallBack_t

```
typedef Bool (*tmalDemuxMpegTSSectionCallBack_t)(
      UInt32   pid,
      UInt8   *section,
      UInt32   sectionLength,
      void    *userData,
      UInt32  *userOutData
);
```

Fields

`pid`	The PID for which this section was found.
`section`	Pointer to the start of the MPEG section (first byte is table ID). Section is CRC'd when the format is **demuxMpegTSSectionCRC**.
`sectionLength`	Length of the section (*not* the **section_length** field as specified by MPEG-2 but the actual number of bytes in the whole section).
`userData`	Pointer passed with the request for redirection of this PID. (This field can be used to store, for instance, section filter data.)
`userOutData`	Pointer to an arbitrary value that you can set. This value is passed in the **userSender** field of the packet that will be sent to the queue. Normally, this field is used to pass on the PID (for sections or data is not filtered with a callback function). You can encode the PID when neccessary.

Description

This function can be passed to the demultiplexer on request of a non-audio/video PID redirection. When the function passed is not Null, the function is called before the section is passed to the queue. With this callback function, the application can implement DVB section filtering.

tmalDemuxMpegTSControlArgs_t

```
typedef struct tmalDemuxMpegTSControlArgs {
   union {
      struct {
         UInt32  pcrPid;
         UInt32  videoPid;
         UInt32  mainAudioPid;
         UInt32  secondaryAudioPid;
         UInt32  index;
      } changePids;
      struct {
         UInt32                                    pid;
         UInt32                                    queueIndex;
         UInt32                                    clockIndex;
         tmalDemuxMpegTSRedirectedOutputFormat_t format;
         tmalDemuxMpegTSSectionCallBack_t          callback;
      } addRedirectedPid;
      struct {
         UInt32 pid;
      } removeRedirectedPid;
   } args;
} tmalDemuxMpegTSControlArgs_t, *ptmalDemuxMpegTSControlArgs_t;
```

Fields

`changePids`

Used when the control command is **TMAL_demuxMpegDEMUXMPEGTS_CHANGE_PIDS**. The fields are set to be set to the requested PID numbers. Each unused PID is to be set to **DEMUXMPEGTS_NO_PID**. Refer also to these functions:

```
tmolDemuxMpegTSChangeVideoPid
tmolDemuxMpegTSChangeMainAudioPid
tmolDemuxMpegTSChangeSecondaryAudioPid
tmolDemuxMpegTSChangePcrPid
tmolDemuxMpegTSChangeToNewPids
```

`addRedirectedPid`

Used when the control command is **TMAL_DEMUXMPEGTS_ADD_PID_REDIRECTION**. The parameters are as follows:

pid: the PID for which extra information is requested.

queueIndex: the index in the queue in which you want the demultiplexer to put the data. Should be less than

DEMUXMPEGTS_MAX_NROF_REQUESTED_QUEUES + the number of default outputs of the demultiplexer.

clockIndex: the clock index for which the offsets must be entered, in the **packet->header->time** fields. See *Outputs* on page 80.

	format: the format in which the extra data is requested. See **tmalDemuxMpegTSRedirectedOutputFormat_t**.
	callBack: when not Null, this field points to callback function that is called (when **format** is **demuxMpegTSSection** or **demuxMpegTSSectionCRC**) before sending the section to the queue.
`removeRedirectedPid`	Used when the control command is **TMAL_DEMUXMPEGTS_REMOVE_REDIRECTED_PID**. **pid** is the previously redirected PID that should no longer be redirected.

Description

This is the data structure used to pass commands from the tmol layer to the tmal layer. You normally call the tmol layer functions, which have a functional interface. This data is then put into the **tmalDemuxMpegTSControl_t** structure which is then passed to the control queue.

These commands can be invoked by calls to the functions:

```
tmolDemuxMpegTSChangeVideoPid
tmolDemuxMpegTSChangeMainAudioPid
tmolDemuxMpegTSChangeSecondaryAudioPid
tmolDemuxMpegTSChangeToNewPids
tmolDemuxMpegTSAddRedirectedPid
tmolDemuxMpegTSRemoveRedirectedPid
```

DemuxMpegTS API Functions

This section presents the DemuxMpegTS component functional interface.

tmolDemuxMpegTSOpen

```
extern tmLibappErr_t tmolDemuxMpegTSOpen(
   Int   *instance
);
```

Parameters

instance	Pointer (returned) to the opened instance.

Return Codes

TMLIBAPP_ERR_MEMALLOC_FAILED	Memory allocation failed.
TMLIBAPP_OK	Success.

The function can also return any code produced by **tsaDefaultOpen**.

Description

Opens an instance of the demultiplexer. (Refer to the documentation on tmol.) It calls **tsaDefaultOpen** and thus indirectly, **tmalDemuxMpegTSOpen**.

The function creates a demultiplexer task with no preemption and no time slicing. This means that the demultiplexer runs until it blocks on one of it input queues.

tmolDemuxMpegTSInstanceSetup

```
extern tmLibappErr_t tmolDemuxMpegTSInstanceSetup(
   Int                                instance,
   ptmolDemuxMpegTSInstanceSetup_t    setup
);
```

Parameters

`instance`	Instance, returned by **tmolDemuxMpegTSOpen**.
`setup`	Pointer to the demultiplexer's setup data structure. See **tmolDemuxMpegTSInstanceSetup_t**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not open. Triggered as an assert.
`TMLIBAPP_ERR_ALREADY_SETUP`	The instance is already set up. Triggered as an assert.
`DEMUXMPEGTS_ERR_INVALID_NROF_BUFFERS`	nrofInputBuffers ≤ 0. Triggered as an assert.
`DEMUXMPEGTS_ERR_INVALID_BUFFER_SIZE`	**inputBufferSize** ≤ 0. Triggered as an assert.
`DEMUXMPEGTS_ERR_INVALID_CLOCK_TICK_VALUE`	**ticksPerSecond** ≤ 0. Triggered as an assert.
`DEMUXMPEGTS_ERR_INVALID_CLOCK_HANDLE`	During instance setup, the demultiplexer was not passed a valid clock handle. Triggered as an assert.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory could not be allocated.
`TMLIBAPP_ERR_NULL_DATAINFUNC`	The datain function is not specified. Triggered as an assert.
`TMLIBAPP_ERR_NULL_DATAOUTFUNC`	The dataout function is not specified. Triggered as an assert.
`TMLIBAPP_ERR_NULL_PROGRESSFUNC`	The progress function is not specified. Triggered as an assert.
`TMLIBAPP_ERR_NULL_ERRORFUNC`	The error function is not specified. Triggered as an assert.
`TMLIBAPP_ERR_NULL_ERRORFUNC`	Either the input or output descriptors are Null. Triggered as an assert.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.

The function can also return any error code produced by **tsaDefaultInstanceSetup**.

Description

Sets up the instance previously opened by **tmolDemuxMpegTSOpen**. Memory is allocated to store runtime instance data. The instance is marked as setup. You should call **tmolDemuxMpegTSInstanceSetup** only once for each instance. The clock handle for index 0 is initialized to a **tsaClock_t** instance running at 90 kHz. This clock will be locked to the tuned program's PCR.

Because the demultiplexer does not copy data, it needs some extra information about its environment: the number of input buffers (used for copying cross input buffer packets), the input buffer size, and OS clock ticks (used for calculating a timeout on the datain function).

All instance variables extracted (video PID etc.) are set to unknown. After a successful call to **tmolDemuxMpegTSInstanceSetup**, the instance is ready to be started.

Note
The clock instance is initialized in this tmolDemuxMpegTSInstanceSetup function. Thus, other components that inspect the same clock, such as the closed captioning decoder or the video and audio decoders, might need to be started later or they should not depend on getting an initialized tsaClock_t instance.

tmolDemuxMpegTSGetInstanceSetup

```
extern tmLibappErr_t tmolDemuxMpegTSInstanceSetup(
   Int                                instance,
   ptmolDemuxMpegTSInstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.
`setup`	Pointer to variable in which to return a pointer to the demultiplexer's setup data structure. See page 105.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.

Description

This function, used during initialization of the demultiplexer, returns the default settings for the demultiplexer instance. The instance setup can then be further initialized by your application, which normally defines all the queues and the progress and error functions and then passes the fully configured setup structure to **tmolDemuxMpegTSInstanceSetup**.

tmolDemuxMpegTSStart

```
extern tmLibappErr_t tmolDemuxMpegTSStart(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_ALREADY_STARTED`	The instance is already started. Triggered by **tmAssert**.

Description

Starts the previously opened and set up instance of the demultiplexer. The function expects that the empty queues of the audio and video outputs contain empty packets. Then the demultiplexer starts waiting for input data from the input queue.

tmolDemuxMpegTSStop

```
extern tmLibappErr_t tmolDemuxMpegTSStop(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by **tmAssert**.

Description

The function calls **tsaDefaultStop**, which stops the demultiplexer. After the demultiplexer stops, its main loop exits. More information on stop functions can be found in the TSSA documentation.

Once stopped, the demultiplexer cannot be set up again.

tmolDemuxMpegTSClose

```
extern tmLibappErr_t tmolDemuxMpegTSClose(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_STOPPED`	The instance is not stopped before closing. Triggered by **tmAssert**.

The function can also return any code produced by **tsaDefaultClose**.

Description

Closes a (stopped) instance of the demultiplexer.

Note
The clock handles created by the demultiplexer are destroyed and can no longer be used by other components.

tmolDemuxMpegTSChangeVideoPid

```
extern tmLibappErr_t tmolDemuxMpegTSChangeVideoPid(
   Int      instance,
   UInt32   newVideoPid,
   UInt32   index
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.
`newVideoPid`	The PID from which the demultiplexer must extract data. This data will go in elementary stream form to the video output of the demultiplexer.
`index`	The relative index from **DEMUXMPEGTS_VIDEO_OUTPUT** of the queue to which this PID should go. (0 ≤ **index** < **DEMUXMPEGTS_NROF_AV_OUTPUTS**)

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by tmAssert.
`DEMUXMPEGTS_ERR_INVALID_QUEUE_INDEX`	The **index** ≥ **DEMUXMPEGTS_NROF_AV_OUTPUTS**.
`DEMUXMPEGTS_ERR_NO_OUTPUTDESCRIPTOR`	No IODescriptor is installed for this **DEMUXMPEGTS_VIDEO_OUTPUT** + **index**.

Description

This function prepares and sends a command to the demultiplexer task, which then synchronously reacts on it. The command is sent with default priority.

Upon receipt of the command, the demultiplexer task stops producing packets from the current PID and starts extracting packets of the requested video PID. If the PIDs are the same, the command has no effect and does not cause any loss of data.

Note
This function may schedule the current task out.

tmolDemuxMpegTSChangeMainAudioPid

```
extern tmLibappErr_t tmolDemuxMpegTSChangeMainAudioPid(
   Int      instance,
   UInt32   newAudioPid,
   UInt32   index
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.
`newAudioPid`	The PID from which the demultiplexer must extract data. This data will go in elementary stream form to the main audio output of the demultiplexer.
`index`	The relative index from **DEMUXMPEGTS_MAIN_AUDIO_OUTPUT** of the queue to which this PID should go. (0 ≤ **index** < **DEMUXMPEGTS_NROF_AV_OUTPUTS**)

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by **tmAssert**.
`DEMUXMPEGTS_ERR_INVALID_QUEUE_INDEX`	The **index** ≥ **DEMUXMPEGTS_NROF_AV_OUTPUTS**.
`DEMUXMPEGTS_ERR_NO_OUTPUTDESCRIPTOR`	No IODescriptor is installed for this **DEMUXMPEGTS_AUDIO_OUTPUT** + **index**.

Description

This function prepares and sends a command to the demultiplexer task, which then synchronously reacts on it. The command is sent with default priority.

Upon receipt of the command, the demultiplexer task stops producing packets from the current PID and starts extracting packets of the requested audio PID. If the PIDs are the same, the command has no effect and will not cause any loss of data.

Note
This function may schedule the current task out.

tmolDemuxMpegTSChangeSecondaryAudioPid

```
extern tmLibappErr_t tmolDemuxMpegTSChangeSecondaryAudioPid(
   Int      instance,
   UInt32   newAudioPid,
   UInt32   index
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.
`newAudioPid`	The PID from which the demultiplexer must extract data. This data will go in elementary stream form to the secondary audio output of the demultiplexer.
`index`	The relative index from **DEMUXMPEGTS_SECONDARY_AUDIO_OUTPUT** of the queue to which this PID should go. (0 ≤ **index** < **DEMUXMPEGTS_NROF_AV_OUTPUTS**)

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by **tmAssert**.
`DEMUXMPEGTS_ERR_INVALID_QUEUE_INDEX`	The index ≥ **DEMUXMPEGTS_NROF_AV_OUTPUTS**.
`DEMUXMPEGTS_ERR_NO_OUTPUTDESCRIPTOR`	No IODescriptor is installed for this **DEMUXMPEGTS_SECONDARY_AUDIO_OUTPUT** + index.

Description

This function prepares and sends a command to the demultiplexer task, which then synchronously reacts on it. The command is sent with default priority.

Upon receipt of the command, the demultiplexer task stops producing packets from the current PID and starts extracting packets of the requested secondary audio PID. If the PIDs are the same, the command has no effect and will not cause any loss of data.

Note
This function may schedule the current task out.

tmolDemuxMpegTSChangePcrPid

```
extern tmLibappErr_t tmolDemuxMpegTSChangePcrPid(
   Int      instance,
   UInt32   newPcrPid,
   UInt32   index
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.
`newPcrPid`	The PID from which the demultiplexer must extract the PCR.
`index`	The relative index from **DEMUXMPEGTS_SECONDARY_AUDIO_OUTPUT** of the queue to which this PID should go. (0 ≤ **index** < **DEMUXMPEGTS_NROF_AV_OUTPUTS**)

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	when the instance is not set up with **tmolDemux-MpegTSInstanceSetup**. Triggered by **tmAssert**.
`DEMUXMPEGTS_ERR_INVALID_QUEUE_INDEX`	The **index** ≥ **DEMUXMPEGTS_NROF_AV_OUTPUTS**.
`DEMUXMPEGTS_ERR_INVALID_CLOCK_HANDLE`	No clock handle was installed for this index at instance setup time.

Description

The PCR belongs to a certain PID and you can select the PID from which the PCR needs to be taken. Normally, the PCR's PID is specified in the PMT for a program. Artifacts can occur when the PCR and the audio/video PID's decoders have no relation.

tmolDemuxMpegTSChangeToNewPids

```
extern tmLibappErr_t tmolDemuxMpegTSChangeToNewPids(
   Int      instance,
   UInt32   newPcrPid,
   UInt32   newVideoPid,
   UInt32   newMainAudioPid,
   UInt32   newSecondaryAudioPid,
   UInt32   index
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.
`newPcrPid`	The PID from which the demultiplexer must extract the PCR.
`newVideoPid`	The PID from which the demultiplexer must extract the video PES.
`newMainAudioPid`	The PID from which the demultiplexer must extract the main audio PES.
`newSecondaryAudioPid`	The PID from which the demultiplexer must extract the secondary audio PES.
`index`	The relative index from **DEMUXMPEGTS_VIDEO_OUTPUT** of the queue to which this PID should go. (0 ≤ **index** < **DEMUXMPEGTS_NROF_AV_OUTPUTS**)

Return Codes

`TMLIBAPP_OK`	Success.

The function can return any code produced by these functions:

```
tmolDemuxMpegTSChangeVideoPid
tmolDemuxMpegTSChangeSecondaryAudioPid
tmolDemuxMpegTSChangeMainAudioPid
tmolDemuxMpegTSChangePcrPid
```

Description

For each of the four input PIDs not set to **DEMUXMPEGTS_NO_PID**, the demultiplexer sets the selected PID for the PCR, video, main audio and secondary audio to the requested PID. The requests are honored whether the PIDs exist in the bitstream or not.

Note
This function may schedule the current task out.

tmolDemuxMpegTSAddRedirectedPid

```
extern tmLibappErr_t tmolDemuxMpegTSAddRedirectedPid(
   Int                                    instance,
   UInt32                                 pid,
   UInt32                                 queueIndex,
   UInt32                                 clockIndex,
   tmalDemuxMpegTSRedirectedOutputFormat_t  format,
   Pointer                                userData,
   tmalDemuxMpegTSSectionCallBack_t       callBack
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.
`pid`	The PID that must be redirected.
`queueIndex`	An index, in the outputDescriptors array, of the demultiplexer's instance. Queues must be attached to this **queueIndex**'s entry. Note that user-requested queues start after the default outputs of the demultiplexer.
`clockIndex`	The index from which the clock offsets must be taken and put into **packet->header->time**. See *Outputs* on page 80 and **tmalDemuxMpegTSConfig_t**.
`format`	Request transport packets, MPEG-2 sections or MPEG-2 sections with CRC. See **tmalDemuxMpegTSRedirected-OutputFormat_t** on page 99.
`userData`	Arbitrary pointer that will be passed back to the application when the callBack function is called.
`callback`	The callback function that, when not Null, is called for each section of this PID before the section is passed to the queue. For arguments to the callback function, refer to the **tmalDemuxMpegTSSectionCallBack_t**. Note that for sections that have the callback function installed, the **userSender** field of the **tmAvPacket_t** is not the regular PID number, but a value that the callBack specifies. It is allowed to call other tmol control functions (such as another **AddRedirectPid** or **RemoveRedirectedPid**, but not the 'close' or 'stop' functions) from within the callback function.

Return Codes

`TMLIBAPP_OK`	Success.
`DEMUXMPEGTS_ERR_INVALID_QUEUE_INDEX`	The queue index exceeds the maximum number of queues that can be used for redirection. See *RC-5 Inputs*

and Outputs on page 60. The maximum number is **DEMUXMPEGTS_MAX_NROF_REQUESTED_QUEUES** + the number of default outputs of the demultiplexer.

`DEMUXMPEGTS_ERR_INVALID_REQUESTED_PID`
The PID was already requested.

`DEMUXMPEGTS_ERR_INVALID_QUEUE_INDEX`
The requirement that 0 ≤ **clockIndex** < **DEMUXMPEGTS_NROF_AV_OUTPUTS** is violated.

`TMLIBAPP_ERR_MEMALLOC_FAILED`
Memory allocation failed.

`TMLIBAPP_ERR_INVALID_INSTANCE`
The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.

`TMLIBAPP_ERR_NOT_SETUP` The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by **tmAssert**.

Description

Dynamically add a redirection of packets of the specified PID. The caller provides the PID and a queue index. This queue index should lie in the outputDescriptors of the demultiplexer's instance and should be a fully initialized InOutDescriptor. (See *Outputs* on page 80. The caller ensures that the empty queue has a sufficient number of packets and that buffers are pre-allocated. Packets are standard **tmAvFormat_t** packets already set by **tmolGetInstanceSetup**.

You can direct multiple PIDs to the same queue index.

A PID cannot be redirected to multiple queues.

You can direct multiple requested PIDs to the same PSI and private data queue indices.

Output can be requested in the form of transport packets (188 bytes) or MPEG-2 sections. Specify the form with the **tmalDemuxMpegTSRedirectedOutputFormat_t** enumeration type. When sections are requested, the pre-allocated buffers should be big enough to handle maximum size sections (or the data will be discarded). An optional CRC is performed on the section. If the CRC does not match, the section is discarded. Optionally, a callback function can be installed. See **tmalDemuxMpegTSSectionCallBack_t**.

For section output, the PID for which the section is redirected is stored in the **userSender** field of the header of the tmAvFormat packet.

Note
This function may schedule the current task out.

tmolDemuxMpegTSRemoveRedirectedPid

```
extern tmLibappErr_t tmolDemuxMpegTSRemoveRedirectedPid(
   Int      instance,
   UInt32   pid
);
```

Parameters

`instance`	Instance, as returned by **tmolDemuxMpegTSOpen**.
`pid`	The PID for which to stop redirection. Your application is responsible for de-allocating the queues when this PID is the last PID to be redirected on this queue's index.

Return Codes

`TMLIBAPP_OK`	Success.
`DEMUXMPEGTS_ERR_PID_NOT_FOUND`	The PID is not known to the demultiplexer and was probably not requested previously by **tmolDemuxMpegTSAddRedirectedPid**.
`DEMUXMPEGTS_ERR_INVALID_REQUESTED_PID`	The PID was not successfully requested with a call to **tmolDemuxMpegTSAddRedirectedPid**.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by **tmAssert**.

Description

Remove a previously redirected PID with a call to **tmolDemuxMpegTSAddRedirectedPid**.

Note
This function may schedule the current task out.

tmalDemuxMpegTSCRCValue

```
extern UInt32 tmalDemuxMpegTSCRCValue(
   UInt8   *packet,
   Int      len
);
```

Parameters

`packet`	Address of the section. Usually, this is a pointer to the table ID.
`len`	Number of bytes to be taken into account for the CRC. Typically, this is `section length + 3 - 4` Section length is stored in the packet. The value 3 is the number of bytes before the section length. The value 4 is the number of bytes in the CRC value itself. See the MPEG-2 standard for more detail on the CRC.

Return Codes

The CRC value of the packet section.

Description

Calculates the CRC value for an MPEG-2 section. Returns the CRC value which is then to be checked against the CRC value stored in the packet.

Chapter 31

MPEG Program Stream Demultiplexer

Note
This component library is not included with the basic TriMedia SDE, but is available as a part of other software packages, under a separate licensing agreement. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

DemuxMpegPS API Overview

The DemuxMpegPS component is a software TSSA Mpeg program stream demultiplexer. It accepts MPEG2 program streams as described in ISO/IEC 13818-1, Recommendation H.222.0 and MPEG1 system streams as described in ISO/IEC 11172-1.

If the stream is a program or a system stream, the DemuxMpegPS component looks for a pack start code and extracts the system clock reference from the pack header. The demultiplexer will report the system reference clock, using the progress function. DemuxMpegPS extracts the stream IDs for the following data types: Mpeg audio stream, private AC3 audio streams, private PCM audio streams, private Subpic streams, and Mpeg video streams. This information is stored in a table, that gets updated each time a new stream ID is found, and a progress report is sent to notify the application, with this table. The application can either pre-determine what elementary streams need to be send to the audio and video decoders, or can select streams during execution, using the information from the progress report. This can be done with the tmalDemuxMpegPSInstanceConfig function. In the latter case the beginning part of the bitstream may be lost for the decoders. After being given the audio and video stream IDs, the demultiplexer looks for corresponding PES start codes, and parses the PES packets. The audio elementary stream with the given stream ID will be sent on the **DEMUXMPEGPS_AUDIO_OUTPUT** queue. The other audio PES will be ignored, since only one stream ID per output queue can be selected. Similarly, the video will go to the **DEMUXMPEGPS_AUDIO_OUTPUT** queue.

The stream IDs reported to the application are:

1. ISO/IEC 13818-2 or ISO/IEC 11172-2 video stream
2. ISO/IEC 13818-3 or ISO/IEC 11172-3 audio stream.
3. Private_Stream_1, if the sub stream ID is an audio stream ID (AC-3 or PCM) DemuxMpegPS reports it as audio stream. If the sub stream ID is a sub-picture, DemuxMpegPS reports it as sub-picture.

Before DemuxMpegPS sends the packet in the output queue, it attached the PTS and DTS information, extracted from the PES header. This timestamp uses the time field of the packet header, and **avhValidTimestamp** will be set. DTS information is passed attached to an empty packet, and applies to the next packet. For the DTS avhValidTimestamp as well as avhValidDts are set.

For the extracted elementary streams, DemuxMpegPS does not copy the data, when sending the packets, but sends a pointer on the data. It is the responsibilities of the downstream components to return the packets fast enough.

Limitations

The application is responsible for reconnecting the downstream components before DemuxMpegPS is told to start sending output to its queues. DemuxMpegPS does not

install formats on its output queues. Also, the formats need to be installed before output is requested.

The application needs to control the system clock. The reason is that DemuxMpegPS is likely to be attached to a file reader, in which case there is no encoder clock to regenerate, and the system runs at a 90KHz clock of the decoder. In these situations, the audio is usually taken as clock master, that is, the audio system determines the clock value, and the clock frequency is constant. Since DemuxMpegPS does not know about any audio system, it leaves this clock regeneration to the application. An example is given in exolDemuxMpegPS. In the case of streaming data, for instance when VdigVIRaw is used to get input, the application needs to regenerate the encoder's 90 kHz clock and likely the frequency of audio and video hardware needs to be adjusted in order to circumvent underruns or overruns in audio.

DemuxMpegPS Inputs and Outputs

Overview

An overview of the inputs and output of the MPEG demultiplexer is depicted in Figure 5. There is one input, which might be an MPEG2 program stream or MPEG1 system stream.

Next to the error and progress reports, there are three stream outputs: the demultiplexed audio elementary stream, the demultiplexed video elementary stream and the sub-picture video elementary stream.

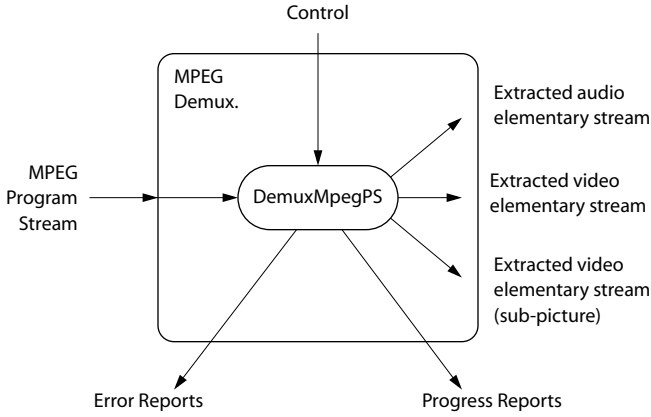

Figure 2 Overview of the Demultiplexer

Inputs

The capability format for the input descriptor is set to

```
tmAvFormat_t tpFormat = {
   sizeof(tmAvFormat_t),              /* size        */
   0,                                 /* hash        */
   0,                                 /* referenceCount */
   avdcSystem,                        /* dataClass   */
   stfMPEG1System | stfMPEG2Program   /* dataType    */
   avdsNone                           /* dataSubtype */
   0                                  /* description */
};
```

The incoming packets are tmAvPackets, which have the format set to one above.

```
#define DEMUXMPEGPS_INPUT    0
```

Outputs

The demultiplexer parses the program stream, decodes the pack header, There are three outputs, the first one being the extracted video elementary stream, which has its capability format set to:

```
tmAvFormat_t videoFormat = {
   sizeof(tmAvFormat_t),   /* size           */
   0,                      /* hash           */
   0,                      /* referenceCount */
   avdcVideo,              /* dataClass      */
   vtfMPEG,                /* dataType       */
   vdfNone,                /* dataSubtype    */
   0                       /* description    */
};
```

The second output is the extracted audio elementary stream, which has its capability format set to:

```
tmAvFormat_t audioFormat = {
   sizeof(tmAvFormat_t),                      /* size           */
   0,                                         /* hash           */
   0,                                         /* referenceCount */
   avdcAudio,                                 /* dataClass      */
   atfAC3 | atfMPEG,                          /* dataType       */
   atfMPEG1_Layer1 | atfMPEG1_Layer2 |
   atfMPEG1_Layer3 | atfMPEG2 | apfGeneric    /* dataSubtype    */
   0                                          /* description    */
};
```

The third output is the video elementary stream for the sub picture. Its capability format is set to:

```
tmAvFormat_t subpicFormat = {
   sizeof(tmAvFormat_t),   /* size           */
   0,                      /* hash           */
   0,                      /* referenceCount */
   avdcVideo,              /* dataClass      */
   vtfMPEG,                /* dataType       */
   vdfNone,                /* dataSubtype    */
   0                       /* description    */
};
```


The output descriptor assignment is:

```
#define DEMUXMPEGPS_VIDEO_OUTPUT  0
#define DEMUXMPEGPS_AUDIO_OUTPUT  1
#define DEMUXMPEGPS_SUBPIC_OUTPUT 2
```

DemuxMpegPS Errors

There is a limited number of error reports produced by DemuxMpegPS. Some reports have the tsaErrorFlagsFatal set which should lead to termination of the instance.

```
tmLibappErr_t
DemuxMpegPSError(Int instId, UInt32 flags, ptsaErrorArgs_t args)
```

DEMUXMPEGPS_ERR_IO_FAILED	Some Dataout or Datain function failed. The error code of the failing OS call may also be reported.

DemuxMpegPS Progress

There is one progress report produced by DemuxMpegPS. When DemuxMpegPS has seen new stream types in the stream, it calls the progress function with the flags set to **DEMUXMPEGPS_STREAM_INFO**. This reports contains the stream information in the description field of the progress argument to the application. The data structure used for the stream information is **tmalDemuxMpegPSStreamInfo_t**.

DemuxMpegPS will also report to the application when it reaches an End Of Stream start code, or an End Of Sequence start code. In that case, the progress function will be called, using the **DEMUXMPEGPS_END_OF_STREAM** flag.

DemuxMpegPS API Data Structures

This section describes the DemuxMpegPS component data structures.

tmolDemuxMpegPSInstanceSetup_t, tmalDemuxMpegPSInstanceSetup_t

```
typedef struct{
   ptsaDefaultInstanceSetup_t      defaultSetup;
   UInt32                          numberOfInputPackets;
   tsaTimeSleepFunc_t              TimSleep;
   tmAudioTypeFormat_t             audio_type;
   tmVideoTypeFormat_t             video_type;
   Bool                            subpic_on;
   UInt32                          audio_stream_id;
   UInt32                          video_stream_id;
   UInt32                          subpic_stream_id;
} tmalDemuxMpegPSInstanceSetup_t, *ptmalDemuxMpegPSInstanceSetup_t;

typedef tmalDemuxMpegPSInstanceSetup_t
        tmolDemuxMpegPSInstanceSetup_t;

typedef ptmalDemuxMpegPSInstanceSetup_t
        ptmolDemuxMpegPSInstanceSetup_t;
```

Fields

`defaultSetup`	See TSSA documentation.
`numberOfInputPackets`	Number of input packets. For buffer management purposes, DemuxMpegPS needs to know the number of input buffers. It cannot be changed on-the-fly.
`TimSleep`	Time sleep function, tmosTimSleep, by default.
`audio_type`	Type of the requested **audio_stream_id**.
`video_type`	Type of the requested **video_stream_id**.
`subpic_on`	When true DemuxMpegPS extracts the requested **subpic_stream_id**.
`audio_stream_id`	Stream ID of the audio stream the application wants DemuxMpegPS to pass to the audio output. **DEMUXMPEGPS_NO_PID** if no stream is requested.
`video_stream_id`	Stream ID of the video stream the application wants DemuxMpegPS to pass to the video output. **DEMUXMPEGPS_NO_PID** if no stream is requested.
`subpic_stream_id`	Stream ID of the sub picture stream the application wants DemuxMpegPS to pass to the sub picture output. **DEMUXMPEGPS_NO_PID** if no stream is requested.

Description

Data structure passed to **tmolDemuxMpegPSInstanceSetup** and **tmalDemuxMpegPS-InstanceSetup** to describe the input and output connections and the initial stream IDs and stream types.

tmolDemuxMpegPSCapabilities_t, tmalDemuxMpegPSCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCaps;
} tmalDemuxMpegPSCapabilities_t, *ptmalDemuxMpegPSCapabilities_t;


typedef tmalDemuxMpegPSCapabilities_t
        tmolDemuxMpegPSCapabilities_t;
typedef ptmalDemuxMpegPSCapabilities_t
        ptmolDemuxMpegPSCapabilities_t;
```

Fields

`defaultCaps`	See TSSA documentation.

tmalDemuxMpegPSCommand_t

```
typedef enum {
    DEMUXMPEGPS_SELECT_MPEG_AUDIO_ID  tsaCmdUserBase + 0x2,
    DEMUXMPEGPS_SELECT_AC3_AUDIO_ID   tsaCmdUserBase + 0x3,
    DEMUXMPEGPS_SELECT_PCM_AUDIO_ID   tsaCmdUserBase + 0x4,
    DEMUXMPEGPS_SELECT_MPEG_VIDEO_ID  tsaCmdUserBase + 0x5,
    DEMUXMPEGPS_SELECT_SUBPIC_ID      tsaCmdUserBase + 0x6,
    DEMUXMPEGPS_REPORT_AUDIO          tsaCmdUserBase + 0x7,
    DEMUXMPEGPS_REPORT_VIDEO          tsaCmdUserBase + 0x8,
    DEMUXMPEGPS_DVD_SUBPIC_ON         tsaCmdUserBase + 0x9,
    DEMUXMPEGPS_DVD_SUBPIC_OFF        tsaCmdUserBase + 0xa,
 } tmalDemuxMpegPSCommand_t;
```

Fields

`DEMUXMPEGPS_SELECT_MPEG_AUDIO_ID`
Set the mpeg audio extracted stream ID to **args–>parameter**.

`DEMUXMPEGPS_SELECT_AC3_AUDIO_ID` Set the ac3 audio extracted stream ID to **args–>parameter**.

`DEMUXMPEGPS_SELECT_PCM_AUDIO_ID` Set the linear PCM audio extracted stream ID to **args–>parameter**.

`DEMUXMPEGPS_SELECT_MPEG_VIDEO_ID`
Set the video extracted stream ID to **args–>parameter**.

`DEMUXMPEGPS_SELECT_SUBPIC_ID` Set the sub picture extracted stream ID to **args–>parameter**.

`DEMUXMPEGPS_REPORT_AUDIO` Report information about the audio stream that is being demultiplexed. DemuxMpegPS will report what type of audio data (Mpeg, Ac3 or PCM), and which stream ID or sub-stream ID is currently selected.

`DEMUXMPEGPS_REPORT_VIDEO` Report which Mpeg video stream is selected.

`DEMUXMPEGPS_DVD_SUBPIC_ON` Not implemented yet.

`DEMUXMPEGPS_DVD_SUBPIC_OFF` By default, this mode is taken.

Description

These commands can be passed as command in a **ptsaControlArgs_t** structure that is passed to **tmolDemuxMpegPSInstanceConfig**. The parameter of the **ptsaControlArgs_t** structure is used to pass the argument, if required. When selecting a stream ID, parameter will contain the value of the selected stream ID.

When you select the **DEMUXMPEGPS_REPORT_AUDIO** or **DEMUXMPEGPS_REPORT_VIDEO** commands, parameter of the **ptsaControlArgs_t** structure is used to return a pointer on a

tmalDemuxMpegPSInfo_t structure, which contains the following information: audio or video data type, and stream ID.

DemuxMpegPS keeps track of the different stream IDs it extracts from the program stream, and stores them in a two-dimensional table. Each time this table gets updated with a new stream ID, DemuxMpegPS reports it to the application, and sends a copy of the table to the application. Sizes of the table are determined by the following two constants:

```
#define DEMUXMPEGPS_NROF_DATATYPES  5
#define DEMUXMPEGPS_NROF_PIDS       4
```

When DemuxMpegPS receives a command from the user to select a specific stream ID, it will check if the stream ID selected by the user is a valid one, but the user is responsible for reconnecting the appropriate downstream components and installing the appropriate formats if necessary.

tmalDemuxMpegPSProgressFlags_t

```
typedef enum {
   DEMUXMPEGPS_STREAM_INFO      0x0001,
   DEMUXMPEGPS_END_OF_STREAM    0x0002,
} tmalDemuxMpegPSProgressFlags_t;
```

Fields

DEMUXMPEGPS_STREAM_INFO	When DemuxMpegPS finds some new stream IDs during the run, it will report it to the application using this progress flag. The stream IDs given at InstanceSetup are not reported.
DEMUXMPEGPS_END_OF_STREAM	When DemuxMpegPS finds an End of Stream start code, or an End of Sequence start code, it will report it to the application using this progress flag.

Description

Used in progress reports, as the progress code.

tmalDemuxMpegPSStreamInfo_t

```
typedef struct {
   tmAudioTypeFormat_t   AudioType,
   tmVideoTypeFormat_t   VideoType,
   Int32                 pid_table[DEMUXMPEGPS_NROF_DATATYPES]
                                  [DEMUXMPEGPS_NROF_PIDS]
} tmalDemuxMpegPSStreamInfo_t;
```

Fields

`AudioType`	Can be **atfNone**, **atfMpeg**, **atfAc3**, **atfLinearPCM** depending on what DemuxMpegPS extracts from the bitstream. Those flags will be OR'ed if the stream contains audio data of different types.
`VideoType`	Can be **vtfMpeg** or **vtfNone**, whether DemuxMpegPS recognizes video data in the stream or not.
`pid_table`	Contains the different stream IDs for the following types of streams: Mpeg audio stream, Mpeg video stream, AC3 private stream, PCM private stream, Subpic private stream. In the case of private streams, since the stream ID is identical for different types of private streams, the sub-stream ID is stored in this table instead of the stream ID.

Description

This data structure is used in **DEMUXMPEGPS_STREAM_INFO** progress report.

tmalDemuxMpegPSInfo_t

```
typedef struct {
   Int32   dataType,
   Int32   streamId
} tmalDemuxMpegPSInfo_t;
```

Fields

`dataType`	Will be set to **atfNone**, **atfMpeg**, **atfAc3** or **atfLinearPCM** depending on the type of audio stream currently selected by DemuxMpegPS. Will be set to **vtfMpeg** if a video stream is selected, **vtfNone** else.
`streamId`	Stream ID or sub-stream ID of the currently selected video or audio stream.

Description

This data structure is used when the user calls **tmalDemuxMpegPSInstanceConfig** with **DEMUXMPEGPS_REPORT_AUDIO** or **DEMUXMPEGPS_REPORT_VIDEO**, set as **args->command**. In return, **args->parameters** will point to a tmalDemuxMpegPSInfo_t structure that contains the information about the audio or video stream currently selected.

DemuxMpegPS API Functions

This section presents the DemuxMpegPS component functional interface.

tmolDemuxMpegPSGetCapabilities, tmalDemuxMpegPSGetCapabilities

```
extern tmLibappErr_t tmolDemuxMpegPSGetCapabilities(
   ptmolDemuxMpegPSCapabilities_t   *cap
);

extern tmLibappErr_t tmalDemuxMpegPSGetCapabilities(
   ptmolDemuxMpegPSCapabilities_t   *cap
);
```

Parameters

`cap`	Pointer to the capabilities structure pointer.

Return Codes

TMLIBAPP_OK	Success.

Description

This function fills in the pointer of a static structure, **tmolDemuxMpegPSCapabilities_t**, **tmalDemuxMpegPSCapabilities_t** maintained by the demultiplexer, to describe the capabilities and requirements of this library.

tmolDemuxMpegPSOpen, tmalDemuxMpegPSOpen

```
extern tmLibappErr_t tmolDemuxMpegPSOpen(
   Int   *instance
);

extern tmLibappErr_t tmalDemuxMpegPSOpen(
   Int   *instance
);
```

Parameters

`instance`	Returned instance.

Return Codes

`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation failed.
`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	When no more instances available.

The function **tmolDemuxMpegPSOpen** can return any code produced by **tsaDefaultOpen**.

Description

Opens an instance of the DemuxMpegPS component. The DemuxMpegPS task is created with preemption. Usually the task should have low priority. The default stack size is set to 4K.

tmolDemuxMpegPSInstanceSetup, tmalDemuxMpegPSInstanceSetup

```
extern tmLibappErr_t tmolDemuxMpegPSInstanceSetup(
   Int                              instance,
   ptmolDemuxMpegPSInstanceSetup_t  setup
);

extern tmLibappErr_t tmalDemuxMpegPSInstanceSetup(
   Int                              instance,
   ptmolDemuxMpegPSInstanceSetup_t  setup
);
```

Parameters

`instance`	Instance previously opened with **tmolDemuxMpegPSOpen**, **tmalDemuxMpegPSOpen**.
`setup`	Pointer to the demultiplexer's setup data structure, see **tmolDemuxMpegPSInstanceSetup_t** and **tmalDemuxMpegPSInstanceSetup_t**.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance opened with **tmolDemuxMpegPSOpen**, **tmalDemuxMpegPSOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_STOPPED`	If the component has not been stopped before calling **tmalVrendVOInstanceSetup.**
`TMLIBAPP_ERR_NOT_OPEN`	when the instance is not opened with **tmolDemuxMpegPSOpen**, **tmalDemuxMpegPSOpen**, triggered via **tmAssert**.
TMLIBAPP_ERR_MEMALLOC_FAILED	No memory could be allocated for the instance.
TMLIBAPP_ERR_INVALID_SETUP	When there is no datainFunc, dataoutFunc, completion Func, errorFunc, progressFunc...
TMLIBAPP_OK	Success.

The function **tmolDemuxMpegPSInstanceSetup** can return any error code produced by **tsaDefaultInstanceSetup**.

Description

The instance previously opened with **tmolDemuxMpegPSOpen** is set up. Memory is allocated for the internally held buffers that are needed for demultiplexing. **tmolDemuxMpegPSInstanceSetup** should be called only once for each instance.

The stream IDs passed in are checked against the IDs given in the MPEG standard. The valid ones are selected as valid stream ID before demultiplexing starts. Elementary stream data of these IDs is immediately extracted.

tmolDemuxMpegPSGetInstanceSetup, tmalDemuxMpegPSGetInstanceSetup

```
extern tmLibappErr_t tmolDemuxMpegPSInstanceSetup(
   Int                               instance,
   ptmolDemuxMpegPSInstanceSetup_t  *setup
);

extern tmLibappErr_t tmalDemuxMpegPSInstanceSetup(
   Int                               instance,
   ptmolDemuxMpegPSInstanceSetup_t  *setup
);
```

Parameters

`instance`	Instance previously opened with **tmolDemuxMpegPSOpen**, **tmalDemuxMpegPSOpen**.
`setup`	Pointer to a pointer to the DemuxMpegPS setup data structure.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolDemuxMpegPSOpen**, **tmalDemuxMpegPSOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmolDemuxMpegPSOpen**, **tmalDemuxMpegPSOpen**, triggered via **tmAssert**.
`TMLIBAPP_OK`	On success.

Description

This function is used during initialization of the decoder. It returns the default settings for the decoder instance. The setup can then be further initialized by the application which normally is filling all the queues and the progress and error functions and then passed to **tmolDemuxMpegPSInstanceSetup**, **tmalDemuxMpegPSInstanceSetup**.

tmolDemuxMpegPSStart, tmalDemuxMpegPSStart

```
extern tmLibappErr_t tmolDemuxMpegPSStart(
   Int   instance
);

extern tmLibappErr_t tmalDemuxMpegPSStart(
   Int   instance
);
```

Parameters

`instance`	Instance previously opened with **tmolDemuxMpegPSOpen**, **tmalDemuxMpegPSOpen**.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolDemuxMpegPSOpen**, **tmalDemuxMpegPSOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	When the instance is not set up with **tmolDemuxMpegTSInstanceSetup**, triggered via **tmAssert**.
`TMLIBAPP_OK`	On success.

Or, in case of **tmolDemuxMpegPSStart**, any error code returned by **tsaDefaultStart**.

Description

The previously opened and set up instance of the decoder is started.

tmolDemuxMpegPSStop, tmalDemuxMpegPSStop

```
extern tmLibappErr_t tmolDemuxMpegPSStop(
   Int   instance
);
```

Parameters

`instance`	Instance previously opened with **tmolDemuxMpegTSOpen**.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolDemuxMpegTSOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmolDemuxMpegTSOpen**, triggered via **tmAssert**.
`TMLIBAPP_OK`	On success.

The function **tmolDemuxMpegPSStop** can return any error code produced by **tsaDefaultStop**.

Description

After a call to Stop, the DemuxMpegPS instance can be restarted via a call to Start. Stop does not free the internally claimed memory.

tmolDemuxMpegPSClose, tmalDemuxMpegPSClose

```
extern tmLibappErr_t tmolDemuxMpegPSClose(
   Int   instance
);

extern tmLibappErr_t tmalDemuxMpegPSClose(
   Int   instance
);
```

Parameters

`instance`	Instance previously opened by **tmolDemuxMpeg-TSOpen**.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolDemuxMpegTSOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_STOPPED`	When the instance is not stopped before, triggered via **tmAssert**.
`TMLIBAPP_OK`	On success.

The function **tmolDemuxMpegPSClose** can return any code produced by **tsaDefaultClose**.

Description

Closes a stopped DemuxMpegPS instance.

tmolDemuxMpegPSInstanceConfig

```
extern UInt32 tmolDemuxMpegPSInstanceConfig(
   Int                  instance,
   UInt32               flags,
   ptsaControlArgs_t    args
);
```

Parameters

`instance`	Instance previously opened with **tmolDemuxMpegPSOpen**.
`flags`	Ignored.
`args`	**args–>command** is one of the command codes from **tmalDemuxMpegPSCommand_t**. When a parameter is required (value of the stream ID the application has selected, for instance), it is passed in **args–>parameter**. **args–>parameter** is also used as a pointer on a **tmalDemuxMpegPSInfo_t** structure, when the information about the video/audio stream currently selected by DemuxMpegPS is asked for by the application.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolDemuxMpegPSOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened.
`TMLIBAPP_ERR_NOT_SETUP`	When the instance is not set up with **tmolDemuxMpegPSInstanceSetup**, triggered via **tmAssert**.
`TMLIBAPP_OK`	Success.

Description

See **tmalDemuxMpegPSCommand_t** for possible control commands.

tmalDemuxMpegPSInstanceConfig

```
extern UInt32 tmalDemuxMpegPSInstanceConfig(
   Int                instance,
   ptsaControlArgs_t  args
);
```

Parameters

`instance`	Instance previously opened with **tmalDemuxMpegPSOpen**.
`args`	**args->command** is one of the command codes from **tmalDemuxMpegPSCommand_t**.When a parameter is required (value of the stream ID the application has selected, for instance), it is passed in via the args structure **args–>parameter**. **args–>parameter** is also used as a pointer on a **tmalDemuxMpegPSInfo_t** structure, when the information about the video/audio stream currently selected by DemuxMpegPS is required by the application.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmalDemuxMpegPSOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened.
`TMLIBAPP_ERR_NOT_SETUP`	When the instance is not set up with **tmalDemuxMpegPSInstanceSetup**, triggered via **tmAssert**.
`TMLIBAPP_OK`	Success.

Description

See **tmalDemuxMpegPSCommand_t** for possible control commands.

Chapter 32

VdigVIRaw API

Note
This component library is not included with the basic TriMedia SDE, but is available as a part of other software packages, under a separate licensing agreement. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

VdigVIRaw API Overview

VdigVIRaw is the TSSA abstraction to the Video-In "Raw" interface as specified in the TriMedia device libraries. For this reason there is only an tmol layer implemented.

A VdigVIRaw module captures raw 8-bit data from the Video-In peripheral and puts this in pre-allocated buffers and sends these buffers to its only output.

VdigVIRaw Inputs and Outputs

Overview

There are no TSSA inputs to a VdigVIRaw component and only a single output. The output contains the full captured packets.

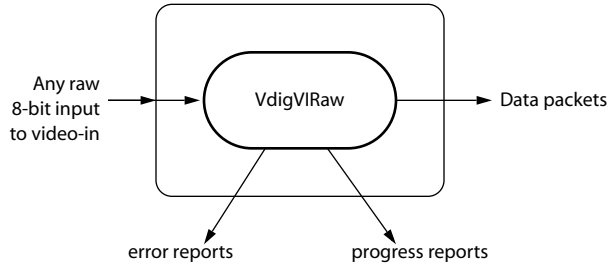

Figure 3 Overview of a VdigVIRaw component

Inputs

There are no TSSA-inputs to a VdigVIRaw module. The input is taken from the Video-In peripheral, thus VdigVIRaw can be seen as a producer of data.

Outputs

The outgoing packets are Video-In captured buffers with raw data, i.e., the Video-In peripheral is operated in the raw 8-bit data mode.

Packets are time-stamped in the Interrupt Service Routine (ISR). The time-stamps can be used for clock recovery purposes in for instance an MPEG-2 system. Since the ISR has an application dependent interrupt latency, the time-stamps are passed through a low pass filter that averages out the interrupt latency variations. In the ISR, the current value of f "cycles" is taken. The previous value recorded in the ISR is stored as start-time-stamp in

the one of the time-stamp fields in the packet. The end-time-stamp is calculated according to:

$$ets = sts + \frac{\sum_{i=0}^{nrofTaps-1} (ets_i - sts_i)}{nrofTaps}$$

where:

ets	end-time-stamp
sts	start-time-stamp
nrofTaps	number of taps (an instance variable determined by the applicaton)

The output packets follow the default tmAvPackets structure and have the following fields set in the header:

`time.hiTicks`	Contains the start time stamp which is the time stamp of the TriMedia clock (cycles custom_op) at the start of buffer capture (or the end of the previous buffer capture).
`time.ticks`	Contains the end time stamp which is the time stamp of the TriMedia cpu clock (cycles custom_op) at the end of the buffer capture. This value takes into account an averaging filter, of which the number of taps can be set. The number of taps is the number of capture times that are averaged.
`buffersInuse`	Set to 1. The VdigVIRaw module does not handle multiple buffers per packet.
`dataSize`	Set to the pre-allocated data size.

There is one output which can be used with the following manifest constant:

```
#define VDIGVIRAW_OUTPUT_ID   0
```

The output format is set to a generic data type, since VdigVIRaw can be used for any data:

```
ststatic tmAvFormat_t Format = {
   sizeof(tmAvFormat_t),   /* size                           */
   0, 0,                   /* hash, referenceCount           */
   avdcGeneric,            /* dataClass                      */
   0, 0, 0                 /* dataType, subtype, description */
};
```

VdigVIRaw Errors

There is one error function that can be invoked by the VdigVIRaw component, and it is invoked when the dataoutFunc returned an error. This usually is a fatal OS error.

The errorFunc is the default TSSA errorFunc and has the following prototype.

```
tmLibappErr_t
VdigVIRawError( Int instId, UInt32 flags, ptsaErrorArgs_t args )
```

VdigVIRaw Progress

There is one progress function from VdigVIRaw, which is invoked when it tries to retrieve a packet from the empty queue but could not get one. At that point, data will be lost since the interrupt service routine cannot block on the empty queue. The progress report is invoked so the application can take appropriate actions.

During instance setup, the application installs the number of buffers that will be lost in a situation like this. This can be useful when more time is needed to recover from the erroneous situation and when the application would rather miss a big gap in its incoming data than a few smaller ones.

The progress function is the default TSSA progress function with the following prototype:

```
tmLibappErr_t
VdigVIRawProgress( Int instId, UInt32 flags, ptsaProgressArgs_t args )
```

VdigVIRaw Configuration

VdigVIRaw cannot be reconfigured. This function is not supported.

VdigVIRaw API Data Structures

This section presents the tmolVdigVIRaw component data structures.

tmolVdigVIRawInstanceSetup_t

```
typedef struct tmolVdigVIRawInstance {
   ptsaDefaultInstanceSetup_t   defaultSetup;
   UInt32                       buffersToLose;
   UInt32                       nrofTaps;
} tmolVdigVIRawInstanceSetup_t, *ptmolVdigVIRawInstanceSetup_t;
```

Fields

`defaultSetup`	See TSSA documentation.
`buffersToLose`	The number of buffers that will be lost when the VdigVIRaw component sees an empty empty-queue. This value can be set higher to 1 when the application needs more time to recover from erroneous situations and rather misses one big block of data than a couple of smaller ones. This is implemented by skipping a number of video-in interrupts.
`nrofTaps`	The number of taps taken to average out the timestamps.

Description

Used by **tmalDemuxMpegTSInstanceSetup_t**.

tmolVdigVIRawCapabilities_t

```
typedef struct tmolVdigVIRawCapabilities{
   ptsaDefaultCapabilities_t    defaultCaps;
} tmolVdigVIRawCapabilities_t, *ptmolVdigVIRawCapabilities_t;
```

Fields

`defaultCaps`	See TSSA documentation.

Description

A VdigVIRaw instance is not re-entrant, since it is an interrupt service routine.

tmolVdigVIRawError_t

```
typedef enum {
   VDIGVIRAW_ERR_INVALID_INTERRUPT_PRIORITY = Err_base_VDigVIRaw+0x01,
   VDIGVIRAW_ERR_BUFFER_ALLOCATION          = Err_base_VDigVIRaw+0x02,
   VDIGVIRAW_ERR_ALIGNMENT                  = Err_base_VDigVIRaw+0x03,
   VDIGVIRAW_ERR_BUFFER_SIZE_ALIGNMENT      = Err_base_VDigVIRaw+0x04,
   VDIGVIRAW_ERR_ALLOCATED_BUFFERS          = Err_base_VDigVIRaw+0x05,
   VDIGVIRAW_ERR_BUFFER_SIZE                = Err_base_VDigVIRaw+0x06,
   VDIGVIRAW_ERR_NOT_ENOUGH_INPUT_BUFFERS   = Err_base_VDigVIRaw+0x07,
   VDIGVIRAW_ERR_INVALID_NROF_TAPS          = Err_base_VDigVIRaw+0x08
} tmolVdigVIRawError_t;
```

Fields

`VDIGVIRAW_INVALID_INTERRUPT_PRIORITY`
: The interrupt priority was not set to a value of type **intPriority_t**.

`VDIGVIRAW_ERR_BUFFER_ALLOCATION`
: One of the empty packets did not have a buffer allocated.

`VDIGVIRAW_ERR_ALIGNMENT`
: One of the empty packets has a buffer allocated that is not cache-aligned This is a video-in peripheral restriction.

`VDIGVIRAW_ERR_BUFFER_SIZE_ALIGNMENT`
: One of the empty packets has a buffer allocated that does not end at a cache-line boundary. This is a video-in peripheral restriction.

`VDIGVIRAW_ERR_ALLOCATED_BUFFERS`
: The VdigVIRaw module can not handle multiple buffers per packet.

`VDIGVIRAW_ERR_BUFFER_SIZE`
: Not all empty buffers have the same size.

`VDIGVIRAW_ERR_NOT_ENOUGH_INPUT_BUFFERS`
: VdigVIRaw requires at least 3 empty packets, two in use by the video-in peripheral and one in use by the component that receives the data from the queue.

`VDIGVIRAW_ERR_INVALID_NROF_TAPS`
: The number of taps for the time-averaging filter is less than 1 or greater than 128.

Description

Enumerates the errors signalled during setup. **Err_base_VdigRIRaw** is **0x20010000**.

tmolVdigVIRawProgress_t

```
typedef enum {
   VDIGVIRAW_LOST_BUFFERS  = 0x0
   VDIGVIRAW_FULL_BUFFER   = 0x1
} tmolVdigVIRawProgress_t;
```

Fields

VDIGVIRAW_LOST_BUFFERS	The VdigVIRaw interrupt handler did not receive any packets from the empty queue. It will lose buffers.
VDIGVIRAW_FULL_BUFFER	One packet has been sent.

Description

Enumerates progress messages.

VdigVIRaw API Functions

This section presents the tmolVdigVIRaw component functional interface.

tmolVdigVIRawGetCapabilities

```
extern tmLibappErr_t tmolVdigVIRawGetCapabilities(
   ptmolVdigVIRawCapabilities_t  *capabilities
);
```

Parameters

`capabilities`	Pointer to a variable in which to return a pointer to the returned capabilities.

Return Codes

The function returns errors from **tmolVdigVIRawGetCapabilitiesM**.

Description

This function calls **tmolVdigVIRawGetCapabilitiesM** for VI unit 0.

tmolVdigVIRawGetCapabilitiesM

```
extern tmLibappErr_t tmolTPInMpeg2GetCapabilitiesM(
   ptmolVdigVIRawCapabilities_t   *capabilities,
   unitSelect_t                    viUnit
);
```

Parameters

`capabilities`	Pointer to a variable in which to return a pointer to the returned capabilities.
`viUnit`	VI unit to get the capabilities for.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function fills in the pointer of a static structure, **tmolVdigVIRawCapabilities_t**, maintained by the library, to describe the capabilities and requirements of this library.

The application can specify for which VI unit it wants to get the capabilities. The library supports up to two VI units.

tmolVdigVIRawOpen

```
extern tmLibappErr_t tmolVdigVIRawOpen(
   Int   *instance
);
```

Parameters

`instance`	Pointer to returned instance.

Return Codes

`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation for the instance parameters failed.
`TMLIBAPP_OK`	Success.

The function can also return any code produced by **tmolDefaultOpen**.

Description

The function calls **tmolVdigVIRawOpenM** to open VI unit 0.

tmolVdigVIRawOpenM

```
extern tmLibappErr_t tmolVdigVIRawOpenM(
   Int           *instance,
   unitSelect_t   viUnit
);
```

Parameters

`instance`	Returned instance. The instance must be used in subsequent API calls.
`viUnit`	The video-in unit to connect.

Return Codes

`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation for the instance parameters failed.
`TMLIBAPP_OK`	Success.

The function can return any code produced by **tmolDefaultOpen** or **viOpenM**.

Description

This function opens an instance of the VdigVIRaw library. The application can specify which VI unit will be opened. The library supports up to two units.

tmolVdigVIRawClose

```
extern tmLibappErr_t tmolVdigVIRawClose(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolVdigVIRawOpen**.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_STOPPED`	The instance is not stopped. Triggered by **tmAssert**.
`TMLIBAPP_OK`	Success.

The function can also return any code from **tmolDefaultClose**.

Description

Closes a stopped instance, frees all memory previously claimed by **tmolDemuxMpegTSOpen** and **tmolDemuxMpegTSInstanceSetup**. It returns the two buffers in use by the video-in peripheral.

tmolVdigVIRawGetInstanceSetup

```
extern tmLibappErr_t tmolVdigVIRawGetInstanceSetup(
   Int                              instance,
   ptmolVdigVIRawInstanceSetup_t  *setup
);
```

Parameters

`instance`	Instance, as returned by **tmolVdigVIRawOpen**.
`setup`	Pointer to variable in which to return the instance setup data structure.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by**tmAssert**.
`TMLIBAPP_OK`	Success.

Description

Returns the default instance parameters.

tmolVdigVIRawInstanceSetup

```
extern tmLibappErr_t tmolVdigVIRawInstanceSetup(
   Int                              instance,
   ptmolVdigVIRawInstanceSetup_t  setup
);
```

Parameters

`instance`	Instance, as returned by **tmolVdigVIRawOpen**.
`setup`	Pointer to the setup data structure.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not opened, triggered as assert.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation for the averaging filter failed.
`VDIGVIRAW_ERR_INVALID_NROF_TAPS`	The number of taps is less than 1, or greater than 128.
`VDIGVIRAW_INVALID_INTERRUPT_PRIORITY`	The interrupt priority is not of type **intPriority_t**.
`ATSC_ERR_INVALID_NROF_BUFFERS`	The value of **nrofInputBuffers** is less than or equal to 0. Triggered as assert.
`ATSC_ERR_INVALID_BUFFER_SIZE`	The value of **inputBufferSize** is less than or equal to 0. Triggered as assert.

The function can also return any error code produced by **tmolDefaultInstanceSetup**, **procGetCapabilities**, **viOpen**, or **viInstanceSetup**.

Description

Sets up the instance and initializes the video-in peripheral.

tmolVdigVIRawStart

```
extern tmLibappErr_t tmolVdigVIRawStart(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolVdigVIRawOpen**.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by **tmAssert**.
`VDIGVIRAW_ERR_BUFFER_ALLOCATION`	One of the packets does not have a pre-allocated buffer.
`VDIGVIRAW_ERR_ALIGNMENT`	One of the packets has a buffer that violates the 64-byte alignment restriction.
`VDIGVIRAW_ERR_ALLOCATED_BUFFERS`	One of the packets has multiple buffers.
`VDIGVIRAW_ERR_BUFFER_SIZE_ALIGNMENT`	The size of the packet is not a multiple of 64 bytes
`VDIGVIRAW_ERR_BUFFER_SIZE`	Not all packets have the same size.
`VDIGVIRAW_ERR_NOT_ENOUGH_INPUT_BUFFERS`	The number of input packets is less than 3.
`TMLIBAPP_OK`	Success.

The function can return any code produced by **tsaDefaultStart**, **viRawSetup**, or **viRawSetup**.

Description

Starts the previously opened and initialized instance. It is expected that the empty queue of the instance contains empty packets. These empty packets are checked against alignment and other restrictions.

tmolVdigVIRawStop

```
extern tmLibappErr_t tmolVdigVIRawStop(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmolVdigVIRawOpen**.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance is not a valid instance opened with **tmolDemuxMpegTSOpen**. Triggered by **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	The instance is not set up with **tmolDemuxMpegTSInstanceSetup**. Triggered by **tmAssert**.
`TMLIBAPP_OK`	On success.

Description

The function calls **tmolDefaultStop**. More information on stop can be found in the TSSA documentation.

After a call to stop, the VdigVIRaw instance cannot be set up again. It can be restarted. When a new instance setup is required, the instance should be closed first.

On stop, there will be two buffers still in use by the video-in peripheral. These are returned only when the instance is closed.

