Book 7—Video Support Libraries

Part B:

Video Processing and Coding

TriMedia

Version 2.0 beta

Table of Contents

Chapter 5 Image Co-Processor (ICP) API

Chapter 6 Variable Length Decoder (VLD) API

Chapter 5

Image Co-Processor (ICP) API

Note
For a general overview of TriMedia device libraries, see Chapter 5, *Device Libraries*, of Book 3, *Software Architecture*, Part A.

Image Co-Processor API Overview

The TriMedia ICP device library provides a set of functions that allow you to access the TriMedia DSPCPU and other peripherals through the TriMedia on-chip data highway. For example, using the ICP device library functions, you can read image data from or write data to SDRAM, or you can write directly to the PCI interface. In particular, this library provides functions to:

- Load filter coefficients from SDRAM.
- Scale and filter image data in horizontal and vertical directions.
- Convert interlaced image data to deinterlaced (progressive scan) image data.
- Perform several different image format conversions, particularly from YUV to RGB.

The TriMedia device libraries are designed to be used to create device drivers. Whereas device drivers are operating-system specific, the device libraries are generic. And whereas device drivers specify a data transfer mechanism, the device libraries gives the data transfer mechanism control to the user.

The example application shows how the ICP device library can be used on its own without a traditional device-driver structure. In a given operating system, it may or may not be useful to create a standard device driver for this peripheral. However, if you decide to create a device driver, the ICP API should be very helpful.

Demonstration Programs

The ICP device library also includes the demonstration program **icptest**, which demonstrates the use of ICP library functions. The source code for this demonstration program serves as an example for developers who want to write their own functions. The program is contained in the example tree of the TriMedia Compilation System.

Using the ICP API

The ICP API is contained in the archived device library, libdev.a. To use the ICP API, you must include the tmICP.h file. The libdev.a device library is linked automatically.

To use the ICP API successfully, you also must do the following:

- Wrap code with the appropriate opening and closing ICP general functions.
- Use the scaling and filtering functions in the code to provide desired functionality.

Figure 1 shows a sequence you will need to follow in order to use the ICP library.

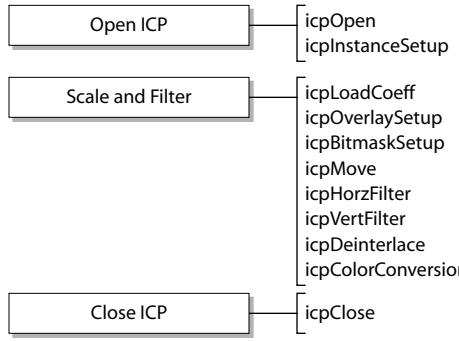


Figure 1 Functions Required for Using the ICP

Limitations

You should be aware of the following hardware and/or software limitations:

- All addresses, or pointers to images, should be 64-bytes aligned, otherwise the results may not be correct.
- For a vertical filter, the source and destination stride, pitch, or line offset must be a multiple of 64 bytes, otherwise the results are incorrect.
- For deinterlacing, the source and destination stride, pitch, or line offset must be a multiple of 64 bytes, otherwise the results are incorrect.
- For a color conversion filter on the 0.5-micron version of the TriMedia processor, if the RGB output is sent to the PCI and the source stride is not a multiple of 64, the output image may have speckles in it. If the source stride is a multiple of 64, the speckles are reduced or completely eliminated. (This problem has been fixed for the 0.35 micron version of the TriMedia processor.)
- The starting address of the microcode must be 128-bytes aligned. This is automatically done in the **icpInstanceSetup** function, which loads and aligns the microcode. If the microcode is 64-bytes aligned, ICP may hang in continuous video.
- There are restrictions on the overlay and output formats when using the **icpColorConversion** function. The overlay image format must be the same type as the output image format generated by the ICP. If the output image format is one of the RGB formats, the overlay must be one of the two RGB overlay formats (**vdfRGB15Alpha** or **vdfRGB24Alpha**). Similarly, if the output image format is YUV, then the overlay must be **vdfYUV422SequenceAlpha**. The formats must be of the same type because the ICP does no conversion of the overlay data. The output/overlay format restrictions are shown in table 15-1 below.

- There are restrictions on the destination PCI address when outputting **vdfRGB24** using the **icpColorConversion** function. The restriction applies when the user wishes to place the output at an address which is offset from the start of the PCI video buffer. The offset must be a multiple of six bytes (i.e., on an even pixel boundary), otherwise the color will be incorrect. For example, consider a PCI video card which has a base address of **0xE0000000**. Valid PCI addresses would be **0xE0000000**, **0xE0000006**, **0xE000000C**, **0xE0000012**, etc.

Output Format	Overlay Format
vdfRGB8A_233	vdfRGB15Alpha or vdfRGB24Alpha
vdfRGB8R_332	
vdfRGB15Alpha	
vdfRGB16	
vdfRGB24	
vdfRGB24Alpha	
vdfYUV422Sequence	vdfYUV422SequenceAlpha
vdfYUV422SequenceAlpha	

Image Co-Processor API Data Structures

The following sections describe the ICP device library data structures.

icpOutputType_t

```
typedef enum {
   icpSDRAM,
   icpPCI
} icpOutputType_t;
```

Description

This enum is used by the struct **icpImageColorConversion_t**. It is used to select between the output being directed to SDRAM bus or the PCI bus.

icpFilterType_t

```
typedef enum {
   icpFILTER,
   icpBYPASS
} icpFilterType_t;
```

Description

This enum is used by the structs **icpImageHorzVert** and **icpImageColorConversion_t**. It is used to specify the filter mode, i.e., whether to use the ICP 5-tap filter or not.

icpImageHorzVert_t

```
typedef struct {
   UInt8            *imageBase;
   Int               inputStride;
   Int               inputHeight;
   Int               inputWidth;
   UInt8            *outputImage;
   Int               outputStride;
   Int               outputHeight;
   Int               outputWidth;
   icpFilterType_t   filterBypass;
   Float             outputPixelOffset;
} icpImageHorzVert_t;
```

Fields

`imageBase`	Pointer to the input image. It should be 64-byte aligned.
`inputStride`	Input image stride.
`inputHeight`	Input image height.
`inputWidth`	Input image width.
`outputImage`	Pointer to the output image.
`outputStride`	Output image stride.
`outputHeight`	Output image height.
`outputWidth`	Output image width.
`filterBypass`	Filtering mode. (Use or bypass the 5-tap filter.)
`outputPixelOffset`	Alignment of the output pixel with respect to the input pixel. This should be in the range of –0.5 to 0.5.

Description

This struct is used by the functions **icpMove**, **icpVertFilter**, **icpHorzFilter and icpDeinterlace**. All the height, width, and stride parameters are assumed to be a positive integer.

icpImageColorConversion_t

```
typedef struct {
   UInt8                  *yBase;
   UInt8                  *uBase;
   UInt8                  *vBase;
   Int                     yInputStride;
   Int                     uvInputStride;
   Int                     inputHeight;
   Int                     inputWidth;
   UInt8                  *outputImage;
   Int                     outputStride;
   Int                     outputHeight;
   Int                     outputWidth;
   icpFilterType_t         filterBypass;
   Float                   outputPixelOffset;
   tmVideoRGBYUVFormat_t   inFormat;
   tmVideoRGBYUVFormat_t   outFormat;
   Bool                    littleEndian;
   Bool                    overlayEnable;
   Bool                    bitMaskEnable;
   icpOutputType_t         outputDestination;
   Float                   alpha0;
   Float                   alpha1;
} icpImageColorConversion_t;
```

Fields

`yBase`	Pointer to the Y image. It should be 64 bytes aligned.
`uBase`	Pointer to the U image. It should be 64 bytes aligned.
`vBase`	Pointer to the V image. It should be 64 bytes aligned.
`yInputStride`	Y image stride.
`uvInputStride`	UV image stride. (U and V assumed to be the same.)
`inputHeight`	Input image height.
`inputWidth`	Input image width.
`outputImage`	Pointer to the output image
`outputStride`	Output image stride.
`outputHeight`	Output image height.
`outputWidth`	Output image width.
`filterBypass`	Filtering mode. (Use or bypass the 5-tap filter.)

`outputPixelOffset`	Alignment of the output pixel with respect to the input pixel. This should be in the range of -0.5 to 0.5.
`inFormat`	Input image format. The available input formats are vdfYUV422planar, vdfYUV420Planar, vdfYUV411Planar, vdf422Interspersed and vdf420Interspersed.
`outFormat`	Output image format. The available output formats are vdfRGB8A_233, vdfRGB8R_332, vdfRGB15Alpha, vdfRGB16, vdfRGB24, vdfRGB24Alpha, vdfYUV422Sequence and vdfYUV422SequenceAlpha.
`littleEndian`	Output data is little endian if TRUE, big endian if FALSE.
`overlayEnable`	Enable the overlay. (See Note 1.)
`bitMaskEnable`	Enable the bitmask. (See Note 2.)
`outputDestination`	The choices are icpPCI or icpSDRAM.
`alpha0`	(See Note 3.)
`alpha1`	(See Note 4.)

Description

This struct is used by the function **icpColorConversion**. All the height, width, and stride parameters are assumed to be a positive integer.

NOTES:

1. This enables the overlay.The user must call icpOverlaySetup before enabling. Once the overlay is set up, overlayEnable can be TRUE or FALSE to display or inhibit display of overlays.

2. This enables the bitmask. The user must call icpBitmaskSetup before enabling. Once the bitmask is set up, bitMaskEnable can be TRUE or FALSE to enable or inhibit the bitmask.

3. The value of alpha0 should be between 0 and 1. alpha0 is chosen if the output is RGB15+alpha or YUV422+alpha and the alpha bit is 0.

4. The value of alpha1 should be between 0 and 1. alpha1 is chosen if the output is RGB15+alpha or YUV422+alpha and the alpha bit is 1.

icpOverlaySetup_t

```
typedef struct {
   UInt8                  *overlayBase;
   Int                     stride;
   Int                     height;
   Int                     width;
   Int                     startX;
   Int                     startY;
   Bool                    littleEndian;
   tmVideoRGBYUVFormat_t   format;
} icpOverlaySetup_t;
```

Fields

`overlayBase`	Pointer to the overlay image.
`stride`	Overlay stride in bytes.
`height`	Overlay height in bytes.
`width`	Overlay width in bytes.
`startX`	Starting pixel for overlay in the original image. It should be between zero and the background image width.
`startY`	Starting line for overlay in the original image. It should be between zero and the background image height.
`format`	Overlay format. Only RGB24+a, RGB15+a and YUV422+a are accepted.
`littleEndian`	Output data is little endian if TRUE, big endian if FALSE.

Description

This struct is used by the functions **icpOverlaySetup** and **icpGetOverlaySetup**. The height, width, and stride parameters are assumed to be a positive integer.

icpBitMaskSetup_t

```
typedef struct {
   UInt8   *bitMaskBase;
   Int      stride;
} icpBitMaskSetup_t;
```

Fields

`bitMaskBase`	Pointer to the bitmask image.
`stride`	Stride of the bitmask image (usually 1/8 of the image width).

Description

This struct is used by the functions **icpBitMaskSetup** and **icpGetBitMaskSetup.**

icpCapabilities_t

```
typedef struct {
   tmVersion_t   version;
   Int           numSupportedInstances;
   Int           numCurrentInstances;
} icpCapabilities_t;
```

Fields

`version`	Returns the library version.
`numSupportedInstances`	Only one instance is supported.
`numCurrentInstances`	Returns the number of open instances.

Description

This struct is used by the function **icpGetCapabilities**

icpInstanceSetup_t

```
typedef struct {
   Bool              reset;
   IntPriority_t     interruptPriority;
   void            (*isr)(void);
} icpInstanceSetup_t;
```

Fields

`reset`	True means to reset the of ICP before installing interrupts.
`interruptPriority`	A value ranging from 0 (lowest priority) to 7 (highest priority).
`isr`	Pointer to the user interrupt service routine (ISR).

Description

This struct is used by the function **icpInstanceSetup.**

Image Co-Processor API Functions

This section presents the ICP device library functions.

icpGetCapabilities

```
tmLibdevErr_t icpGetCapabilities(
   picpCapabilities_t   *icpCap
);
```

Parameters

`icpCap`	Pointer to a variable in which to return a pointer to capabilities data.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NULL_PARAMETER`	Pointer to the struct is NULL.

Description

This function fills in the value of a user-supplied pointer variable which will then point to the single shared capabilities structure for the ICP device library.

icpOpen

```
tmLibdevErr_t icpOpen(
   Int   *instance
);
```

Parameters

`instance`	Instance value.

Return Codes

`TMLIBDEV_OK`	Success.
`ICP_ERR_NO_MORE_INSTANCES`	Returned if the ICP cannot allocate more instances.

Description

This function assigns an instance for usage and resets the ICP with the **icpReset** macro. Note that ICP is a single-instance device.

icpInstanceSetup

```
tmLibdevErr_t icpInstanceSetup(
   Int                  instance,
   icpInstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance value.
`setup`	Pointer to a struct of type **icpInstanceSetup_t** containing setup data.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Some other process already has the device control. The debug version of the device library can assert this.
`ICP_ERR_LOAD_MICRO_CODE`	Could not load the micro code. Check to see if you have the correct microcode file.
Others	tmInterrupts errors.

Description

This function will prepare the ICP for operation by loading the micro code. It will enable or disable the ICP interrupt according to the value of the pointer of the interrupt service routine with **icpEnableINTERRUPT** or **icpDisableINTERRUPT** macros.

It requires that the function **icpOpen** has been called first. It can then be called more than once to re-install the interrupt handler or change the interrupt service routine (ISR).

It also sets up the little-endian or big-endian mode. The mode defaults to match the compiler switch.

icpClose

```
tmLibdevErr_t icpClose(
   Int    instance
);
```

Parameters

`instance`	The instance.

Return Codes

`TMLIBDEV_OK`	Success.
`ICP_ERR_BUSY`	Some other process is using the ICP.
`TMLIBDEV_ERR_NOT_OWNER`	The instance does not match the owner.

Description

The **icpClose** function closes the hardware and deinstalls the interrupt handler.

This resets the ICP (with the **icpReset** macro), close the intICP interrupt with **intClose**, and disables ICP interrupts with the **icpDisableINTERRUPT** macro.

After closing the ICP, you should call **icpOpen** in order to use the ICP filters again.

icpLoadCoeff

```
tmLibdevErr_t icpLoadCoeff(
   Int      instance,
   Int16   *icpFilterCoeff
);
```

Parameters

`instance`	Instance value.
`icpFilterCoeff`	Pointer to a coefficient table stored as Int16.

Return Codes

`TMLIBDEV_OK`	Success.
`ICP_ERR_COEFF_TABLE`	Filter coefficients are outside the range [–512, 511].

Description

The **icpLoadCoeff** function loads the ICP with filter coefficients. If the input pointer is NULL, then it will load the standard coefficients. This function is called only when changing coefficients and it may be called at any time during ICP processing to load a new set of coefficients. It starts the ICP and returns after the coefficients are loaded. The function assumes the following:

- Microcode is already loaded
- All other ICP controls are set and checked outside.

Following successful completion, the coefficient table is loaded in SDRAM and is ready to be used by the ICP filters.

icpMove

```
tmLibdevErr_t icpMove(
   Int                  instance,
   icpImageHorzVert_t   *image
);
```

Parameters

`instance`	Instance value.
`image`	Pointer to the struct containing image data and pointers.

Return Codes

`TMLIBDEV_OK`	Operation successfully completed.
`TMLIBDEV_ERR_NOT_OWNER`	Returned if instance does not match owner.
`ICP_ERR_SETUP_REQUIRED`	ICP has not been set up. Use **icpInstanceSetup**.
`ICP_ERR_INVALID_HEIGHT`	Height not specified correctly.
`ICP_ERR_INVALID_WIDTH`	Width not specified correctly.
`ICP_ERR_INVALID_STRIDE`	Stride not specified correctly.
`ICP_ERR_ADDRESS_NOT_64_ALIGNED`	Source or image address is not 64-bytes aligned.
`ICP_ERR_BUSY`	Attempt to call **icpMove** while the ICP is busy processing another task previously assigned by the same user.

Description

The **icpMove** function moves an image from the area in SDRAM specified by ***inputImage** to the area in SDRAM specified by ***outputImage**.

The function assumes the following:

- Microcode is already loaded.
- ICP is initialized and open.
- The source image is copied back in memory.

icpVertFilter

```
tmLibdevErr_t icpVertFilter(
   Int                  instance,
   icpImageHorzVert_t   *image
);
```

Parameters

`instance`	Instance value.
`image`	Pointer to the struct containing image data and pointers.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned if instance does not match owner.
`ICP_ERR_SETUP_REQUIRED`	ICP has not been set up. Use **icpInstanceSetup**.
`ICP_ERR_INVALID_HEIGHT`	Height not specified correctly.
`ICP_ERR_INVALID_WIDTH`	Width not specified correctly.
`ICP_ERR_INVALID_STRIDE`	Stride not specified correctly.
`ICP_ERR_ADDRESS_NOT_64_ALIGNED`	Source or image address is not 64-bytes aligned.
`ICP_ERR_BUSY`	Attempt to call **icpVertFilter** while ICP is busy processing another task previously assigned by the same user.
`ICP_ERR_VERT_STRIDE_NOT_64_ALIGNED`	Source or destination stride is not a multiple of 64 bytes.

Description

The function **icpVertFilter** vertically filters and scales an image in SDRAM pointed by *inputImage and stores the new image in SDRAM to which **outputImage** points. For vertical filtering, output and input strides should be a multiple of 64, otherwise the output is not guaranteed to be correct.

The function assumes the following:

- Microcode is already loaded (automatically done in **icpOpen**).
- Filter coefficients are already loaded.
- ICP is initialized and open.
- Source image is copied back in memory.

icpHorzFilter

```
tmLibdevErr_t icpHorzFilter(
   Int                  instance,
   icpImageHorzVert_t   *image
);
```

Parameters

`instance`	Instance value.
`image`	Pointer to the struct containing image data and pointers.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned if instance does not match owner.
`ICP_ERR_SETUP_REQUIRED`	ICP has not been set up. Use **icpInstanceSetup**.
`ICP_ERR_INVALID_HEIGHT`	Height not specified correctly.
`ICP_ERR_INVALID_WIDTH`	Width not specified correctly.
`ICP_ERR_INVALID_STRIDE`	Stride not specified correctly.
`ICP_ERR_ADDRESS_NOT_64_ALIGNED`	Source or image address is not 64-bytes aligned.
`ICP_ERR_BUSY`	Attempt to call **icpHorzFilter** while ICP is busy processing another task previously assigned by the same user.

Description

The **icpHorzFilter** function filters an image in SDRAM pointed to by ***inputImage** and stores the new image in SDRAM to which **destImage** points.

The function assumes the following:

- Microcode is already loaded.
- Filter coefficients are already loaded.
- ICP is initialized and open.
- Source image is copied back in memory.

icpDeinterlace

```
tmLibdevErr_t icpDeinterlace(
   Int                  instance,
   icpImageHorzVert_t   *image
);
```

Parameters

`instance`	The instance.
`image`	Pointer to the struct containing image data and pointers.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned if instance does not match owner.
ICP_ERR_SETUP_REQUIRED	ICP has not been set up. Use **icpInstanceSetup**.
`ICP_ERR_INVALID_HEIGHT`	Height not specified correctly.
`ICP_ERR_INVALID_WIDTH`	Width not specified correctly.
`ICP_ERR_INVALID_STRIDE`	Stride not specified correctly.
`ICP_ERR_ADDRESS_NOT_64_ALIGNED`	Source or image address is not 64-bytes aligned.
`ICP_ERR_BUSY`	Attempt to call **icpVertFilter** while ICP is busy processing another task previously assigned by the same user.
`ICP_ERR_VERT_STRIDE_NOT_64_ALIGNED`	Source or destination stride is not a multiple of 64 bytes.

Description

The function **icpDeinterlace** performs interlaced to deinterlaced (progressive scan) conversion of an image in SDRAM pointed by inputImage and stores the new image in SDRAM to which **outputImage** points. For deinterlace filtering, output and input strides should be a multiple of 64, otherwise the output is not guaranteed to be correct. The function requires special filter coefficients to be loaded; these are provided in the device library and have the name **icpDeinterlaceCoeff**. The coefficients are loaded using the **icpLoadCoeff** function; an example being

```
icpLoadCoeff( instance, icpDeinterlaceCoeff );
```

The function assumes the following:

- Microcode is already loaded (automatically done in **icpOpen**).
- Deinterlace filter coefficients are already loaded.
- The ICP is initialized and open.
- Source image is copied back in memory.

icpColorConversion

```
tmLibdevErr_t icpColorConversion(
   Int                          instance,
   icpImageColorConversion_t   *image
);
```

Parameters

`instance`	Instance value.
`image`	Pointer to the struct containing image data and pointers.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned if instance does not match owner.
`ICP_ERR_SETUP_REQUIRED`	ICP has not been set up. Use **icpInstanceSetup**.
`ICP_ERR_INVALID_HEIGHT`	Source or destination height <= 0.
`ICP_ERR_INVALID_WIDTH`	Source or destination width <=0.
`ICP_ERR_INVALID_STRIDE`	Source or destination stride <= 0.
`ICP_ERR_ADDRESS_NOT_64_ALIGNED`	Address pointer must be aligned at a 64 byte block boundary.
`ICP_ERR_BUSY`	Attempt to call **icpColorConversion** while ICP is busy processing.
`ICP_ERR_IMAGE_TYPE`	Requested image format is not supported.
`ICP_ERR_OUTPUT_NOT_DEFINED`	Requested output is neither PCI nor SDRAM.
`ICP_ERR_CANT_WRITE_TO_BIUCTRL`	For PCI output, BIU control register could not be modified.
`ICP_ERR_INVALID_ALPHA`	alpha is outside the range [0,1].
`ICP_ERR_SETUP_OVERLAY_REQUIRED`	Overlay mode requested through **overlayEnable** before **icpOverlaySetup** was called.
`ICP_ERR_SETUP_BITMASK_REQUIRED`	Bitmask mode requested before the function **icpBitMaskSetup** was called.

Description

This function will convert YUV422 planar, YUV420 planar, YUV 411 planar, YUV 420 interspersed and YUV 422 interspersed to RGB 24 + alpha, RGB 24 packed, RGB 16, RGB 15 + alpha, RGB 8A, RGB 8R, YUV 422 sequence, or YUV 422 sequence + alpha formats and send the data to either the **PCI or** SDRAM with or without an overlay or bitmask. Note that when the output is directed to the SDRAM, the ICP does not use overlay and bitmask information.

Required conditions are:

- Images are already copied back.
- Filter coefficients are already loaded.
- Micro code is already loaded in SDRAM.
- ICP is already set up and open.

icpOverlaySetup

```
tmLibdevErr_t icpOverlaySetup(
   Int                  instance,
   icpOverlaySetup_t   *overlay
);
```

Parameters

instance	The instance.
Overlay	Pointer to the overlay information.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NOT_OWNER	Instance does not match owner.
ICP_ERR_BUSY	ICP is still processing some previous job.
ICP_ERR_SETUP_REQUIRED	ICP has not been set up.
ICP_ERR_INVALID_START_VALUE	Invalid **startX** or **startY**.
ICP_ERR_OVERLAY_MODE	Requested mode is not supported.
TMLIBDEV_ERR_NULL_PARAMETER	Pointer to the overlay struct is NULL.

Description

This function sets up an overlay image. If an image overlay is desired then it must be called before calling the **icpColorConversion** function and setting **overlayEnable** to TRUE. **icpColorConversion** can then be called anytime to suppress display of the overlay image by setting the enableOverlay flag to FALSE.

icpGetOverlaySetup

```
tmLibdevErr_t icpGetOverlaySetup(
   Int                  instance,
   icpOverlaySetup_t   *overlay
);
```

Parameters

`instance`	Instance value.
`overlay`	Pointer to the overlay information.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned when instance does not match owner.

Description

Retrieves the current overlay information.

icpBitMaskSetup

```
tmLibdevErr_t icpBitMaskSetup(
   Int                  instance,
   icpBitMaskSetup_t   *bitmask
);
```

Parameters

`instance`	Instance value.
`bitmask`	Pointer to the struct containing bitmask information.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned when instance does not match owner.
`ICP_ERR_SETUP_REQUIRED`	ICP has not been set up.
`TMLIBDEV_ERR_NULL_PARAMETER`	Pointer to the overlay struct is NULL.

Description

This function sets up a bitmask image. If an bitmask image is desired then it must be called before calling the **icpColorConversion** function and setting the bitmaskEnable to TRUE. **icpColorConversion** can then be called anytime to suppress display of the bitmask image by setting the enableBitmask flag to FALSE.

icpGetBitMaskSetup

```
tmLibdevErr_t icpGetBitMaskSetup(
   Int                  instance,
   icpBitMaskSetup_t   *bitmask
);
```

Parameters

`instance`	The instance.
`bitmask`	Pointer to the structure which will receive bitmask information.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned when instance does not match owner.

Description

Retrieves the current bitmask information.

Chapter 6

Variable Length Decoder (VLD) API

Note
For a general overview of TriMedia device libraries, see Chapter 5, *Device Libraries*, of Book 3, *Software Architecture*, Part A.

Introduction

The Variable Length Decoder (VLD) is a co-processor to the TriMedia DSPCPU which assumes responsibility for the Huffman (Entropy) decoding process in MPEG video. Provided with a pointer to an MPEG 1 or MPEG 2 stream as well as some configuration information, it produces as output the macroblock header information and DCT coefficients on a macroblock level. These data are sent to separate buffers located in SDRAM via DMA, and are accessed by the DSPCPU to complete the video decoding process. In general, MMIO registers are used to communicate, control and synchronize for VLD operations.

VLD Operation

The VLD coprocessor supports five major operations:

1. Shift the bit stream
2. Search for the next start code
3. Parse macroblocks
4. Flush the output FIFO
5. Reset the VLD.

The VLD decodes Huffman codes in hardware enabling asynchronous operation with the DSPCPU. The VLD outputs to two buffers in SDRAM, one buffer for the macroblock headers and another for run length encoded DCT coefficients.

VLD Basics

The VLD API functions and data structure definitions are based upon the operations provided by the VLD. The function **vldParseMacroblocks** is used to parse a number of macroblocks. This function should be called after the necessary data such as picture information has been provided using the vldSetPictureInfo function. Once the VLD has started parsing macroblocks or shifting the bitstream, it may stop for any one of the flowing reasons:

- the command completed without exceptions
- a start code was detected
- an error was encountered in the bit stream
- the VLD input DMA completed and the VLD is stalled waiting for more input bitstream data
- one of the VLD output DMAs completed and the VLD is stalled because the output FIFO is full.

Under normal circumstances the DSPCPU can be interrupted whenever the VLD halts. The function **vldGetBits** is provided to get a specific number of bits by shifting the bit-

stream. Other functions that can be used to hand decode the input stream include **vldShowBits** to look at the next bits without shifting the input stream, **vldFlushBits** to skip a number of bits and **vldNextStartCode** to skip bits until a new startcode is encountered.

Macroblock Headers

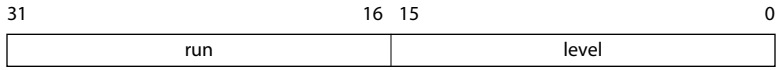
The buffer that is filled with macroblock header information has to be allocated by the user. Its location and size must be given to the VLD before it can start decoding macroblocks. Whenever the VLD notifies the user that the buffer is full, the user must point the VLD to a new buffer. The format of the macroblock header returned by the VLD is described in the data book (Chapter 14, figure 2). In the case of an MPEG-1 stream, the macroblock header takes four 32-bit words, while for an MPEG-2 stream, the buffer is filled with blocks of six 32-bit words; the difference being the second motion vectors. The VLD library provides an efficient function **vldGetMBHeader** to store these four- or six-word blocks into a C structure, **vldTMBHField_t,** for more convenient access and manipulation.

DCT Coefficients

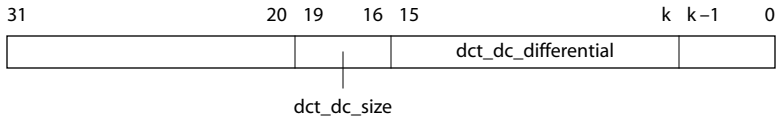
Similar to the buffer requirement for the macroblock headers, the buffer for the DCT coefficients must be allocated explicitly by the user and its location and size passed to the VLD.

The DCT coefficients are represented with 32 bit words. However, there are three formats in which the coefficients can be stored:

- AC coefficients. AC coefficients are stored in the buffer in the following format:

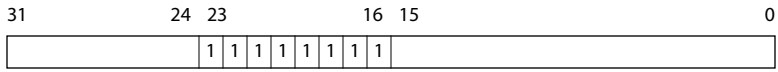


- DC coefficients. The first word of an intra block describes the DC coefficients in the following format:



  Where k = 16 – dct_dc_size bits, bits k-0 are not significant.

- End of block markers. The end of block markers are stored in the buffer in the following format:



Note
The output is generated in sections of 64 bytes, so when the VLD has completed parsing macroblocks, and the last macroblock headers or coefficients do not add up to 64 bytes, the user must flush the hardware buffers explicitly, using **vldFlushOutput**. As flushing is also performed in sections of 64 bytes, some erroneous information will be written into the buffers.

Manipulating the Input Stream

Five functions are available in the VLD library to parse or manipulate the input stream. By using these functions, the input buffer may become empty. This will be notified through a bit in the VLD status register, or by an interrupt if enabled. Three of these functions are provided to manipulate the input bit stream on a per-bit basis; these are **vldGetBits, vldShowBits**, and **vldFlushBits**.

- **vldParseMacroblocks**

  The function **vldParseMacroblocks** is used to instruct the VLD to start decoding the bitstream and fill two buffers with macroblock headers and DCT coefficients. The arguments of this function inform the VLD how many macroblocks to decode and where and how large the output buffers are.

  The function is asynchronous; it will start the VLD and return immediately. The VLD will set the status MMIO register or raise an interrupt in case the requested number of macroblocks is successfully parsed, or when it cannot finish; this may be because the input buffer is empty, one of the output buffers is full, a start code was encountered or a bitstream error occurred.

  If the VLD is halted because the input buffer is empty or one of the output buffers is full, the VLD can be supplied with a new buffer to enable it continue automatically.

- **vldNextStartCode**

  The function **vldNextStartCode** can be called to find the next start code in the input stream. On success, the function returns the start code so the user can determine what specific start code was encountered.

- **vldGetBits**

  The function **vldGetBits** will shift the bit stream a maximum of 32 bits, and return those bits through a pointer.

- **vldShowBits**

  The function **vldShowBits** also returns a requested number of bits from the input stream to the user, but it does not shift the input stream, so it can only show the bits within the shift register.

- **vldFlushBits**

  The function **vldFlushBits** can be used to shift and remove any number of bits from the input stream.

Reset VLD

Calling **vldFlushOutput** ensures that all output buffers are up to date with respect to the current parsed bit stream and that the hardware output buffers have been flushed. The function **vldReset** is provided to reset the VLD state machine and ensure all VLD MMIO registers are in a clean state for parsing an input stream.

Setup for VLD

The function **vldInstanceSetup must be called to set up the VLD library**. This function takes as an argument a struct type **vldInstanceSetup_t** that contains information required by the library. One of the fields, **vldEmptyFunc**, provides the library with a pointer to a callback function that provides the VLD with a filled input buffer. The callback function can be called from the library, an application or an ISR when the input buffer becomes empty. The function should accept a flag as parameter that tells the function whether it is allowed to block until it actually provided the VLD with a new buffer, because there are synchronous commands such as vldGetBits, vldFlushBits and asynchronous command like vldParseMacroblocks in the VLD library.

Other fields of the structure hold the interrupt service routine, the interrupt mask and the interrupt priority for the VLD interrupt. Providing the VLD with new input can take longer than one might actually want to spend in a ISR, a flag is used to remember that the input buffer is empty. The flag is used in the library to determine whether it needs more input before parsing the bitstream. If the flag is set, it will call **vldEmptyFunc**. The macros **vldGetEmptyFlag** and **vldSetEmptyFlag** can be used to access the flag.

In order for the VLD to operate correctly, the following information must be provided:

- Location of the input buffers

  The location and size of the input buffer can be set by a call to the function **vldInput**. The size of the input buffer is passed in bytes.

- Location of the output buffers

  The locations and size of the output buffers are given as arguments to the function **vldParseMacroblock**. In case one of the output buffers is full before the VLD completes parsing, a new location and size can be given by using the macros **vldSetMBH_ADR** and **vldSetMBH_CNT**, or **vldSetRL_ADR** and **vldSetRL_CNT**. These macros will probably be used in the **vldEmptyFunc** as described in the previous paragraph. The size of the output buffers is to be given in 32-bit words. The addresses of the output buffers must be aligned on a 64-byte boundary.

- Information stored in the picture and slice headers

  The slice and picture headers contain information required by the VLD to parse the macroblocks correctly. Therefore, the VLD needs to be informed after a slice or picture header is parsed by a decoder. The information in a picture header can be communicated by calling the function **vldSetPictureInfo**. The only argument of this function is a structure type **vldPictureInfo_t** that holds all information required by the

VLD; for example, **picture_type** and **frame_pred_frame_dct**. Some of the fields are only relevant when an MPEG 2 stream is parsed. The only information in a slice header that is important to the correct operation of the VLD is the quantiser scale which can be set using the macro **vldSetQS** after a slice header is parsed.

Getting Status Information From VLD

As mentioned previously, the VLD halts on certain conditions. These conditions are reflected by bits in the VLD status register that can be checked by the user. You can also instruct the VLD to raise an interrupt on these conditions, by calling **vldInstanceSetup** with the correct interrupt mask. There are six conditions that halt the VLD:

1. The command is completed successfully (bit **SUCCESS**)
2. A startcode in the bitstream is found (bit **STARTCODE**)
3. A syntax error is found in the bitstream (bit **ERROR**)
4. The input buffer has become empty (bit **DMA_IN_DONE**)
5. The run length encoded DCT coefficients output buffer is full (bit **RL_OUTPUT_DONE**)
6. The output buffer for the macroblock headers is full (bit **MBH_OUTPUT_DONE**)

You can check, by polling, whether a start code has been encountered using two methods. The first is to use the macro **vldCheckSTATUS_STARTCODE**. The second is to use **vldGetSTATUS**, which is OR'd with **VLD_STATUS_STARTCODE**.

Note
The **XXX_OUTPUT_DONE** bits are set before the success bit, so the output buffers need to be larger than the actual output for the VLD to complete successfully. Otherwise, the VLD will complain about a full output buffer before it can inform the user that the parse command has been successfully completed.

The MMIO registers that are set to inform the VLD about the locations and sizes of the input and output buffers are kept up to date by the VLD. They continuously reflect the location where the next read and writes are to be done, and how many reads or writes are still left in the current buffers. The macro **vldGetBIT_ADR** can be used to get the byte address in the input buffer from which the VLD is reading; whereas **vldGetMBH_ADR** and **vldGetRL_MBH** can be used to get the (64-byte-aligned) locations where the VLD will write its next output. The macro **vldGetBIT_CNT** can be used to get the number of bytes that can still be read from the input buffer. The macros **vldGetMBH_CNT** and **vldGetRL_CNT** can be used to get the number of 32-bit words that can still be written to the output buffer.

Note
The macros return the requested numbers only, while some of the MMIO registers hold two values in each 32 bit register.

VLD Multiple Streams (Instances) Decoding

For advanced VLD users, multiple stream or instance decoding functionality and data structures are provided in the VLD library. There are two data structures for multiple stream decoding, **vldContext_t** and **vldInstanceInfo_t,** and two functions specifically for context switching in multiple stream decoding. The maximum number of multiple streams or instances in the VLD library is six.

The VLD context consists of the contents of several VLD registers (**VLD_BIT_ADR**, **VLD_BIT_CNT**, **VLD_CTL**, **VLD_IMASK**) and the unused data portion of the VLD input FIFO content up to a 64-byte boundary. The information of the saved input FIFO data consists of the bit offset from a byte boundary, the byte number from the current byte to a 64-byte boundary and the actual buffer data. The context switch should be made at the place when the information is known.

The VLD instance information includes the instance usage flag, the MPEG type flag, the picture information content, instance setup content and instance context. Each VLD instance represents a complete set of the VLD functionality and data utility for each input bitstream.

VLD Example Program

The example program vldtest parses an MPEG-1 or MPEG-2 stream, and compares the VLD output with two reference files to check the correctness of the library. It provides example code on how to set up the VLD initially and an ISR to handle an empty input buffer. It also shows how to parse a picture header or a slice header by hand and pass the important information from these headers to the VLD. The test program also shows how cache coherency can be ensured, as described in *Cache Coherency* in the TriMedia data book.

VLD API Data Structures

This section presents the VLD device library data structures. These data structures are defined in the tmVLD.h header file.

pFnVldEmpty_t

```
typedef void (*pFnVldEmpty_t)(
   Int32   flag
);
```

Fields

`flag`	The function is allowed to block if **flag** is set.

Description

Callback routine which is executed when a function in the VLD library needs more input data. If a full buffer is available at the time of the call, it passes it to the VLD by calling **vldInput**. If input data is not available at the time of the call then there are two possibilities:

1. If flag is set, it will wait until data is available, and then supply the buffer to the VLD.

2. If flag is not set, it should return immediately. Before it returns, it should indicate whether new input is actually provided by calling **vldSetEmptyFlag**.

pFnVldISR_t

```
typedef void (*pFnVldISR_t)( void );
```

Description

VLD interrupt service routine provided by the user to handle those conditions on which the VLD halts and are signalled through an interrupt.

vldCapabilities_t

```
typedef struct {
   tmVersion_t   version;
   Int32         numSupportedInstances;
   Int32         numCurrentInstances;
} vldCapabilities_t, *pvldCapabilities_t;
```

Fields

`version`	Version of this device library.
`numSupportedInstances`	Number of users that can access the VLD device simultaneously.
`numCurrentInstances`	Number of current users.

Description

Used by the function **vldGetCapabilities**.

vldPictureInfo_t

```
typedef struct {
    Uint8   PictureType;
    Uint8   PictureStruct;
    Uint8   FramePFrameD;
    Uint8   intraVLC;
    Uint8   concealMV;
    Uint8   mpeg2Mode;
    Uint8   h_Forw_RSize;
    Uint8   v_Forw_RSize;
    Uint8   h_Back_RSize;
    Uint8   v_Back_RSize;
} vldPictureInfo_t;
```

Fields

`PictureType`	See **picture_coding_type** [6.3.9].
`PictureStruct`	See **picture_structure** [6.3.9].
`FramePFrameD`	See **frame_pred_frame_dct** (MPEG 2) [6.3.10].
`intraVLC`	See **intra_vlc_format** (MPEG 2) [6.3.10].
`concealMV`	See **concealment_motion_vectors** (MPEG 2) [6.3.10].
`mpeg2Mode`	0 if MPEG 1 and 1 if MPEG 2.
`h_Forw_RSize`	See **r_size[0][0]** (MPEG 2) [6.3.17].
`v_Forw_RSize`	See **r_size[0][1]** (MPEG 2) [6.3.17].
`h_Back_RSize`	See **r_size[1][0]** (MPEG 2) [6.3.17].
`v_Back_RSize`	See **r_size[1][1]** (MPEG 2) [6.3.17].

Description

Passed to the VLD library that is using **SetPictureInfo** to inform the VLD hardware about how to decode the incoming bitstream. As indicated, some of the fields only will appear in an MPEG-2 stream and do not need to be set for an MPEG-1 stream.

For more information on the interpretation of the fields, refer to official MPEG standard: the applicable paragraphs in ISO/IEC draft 13818-2 are included in the above comments.

vldMVector_t

```
typedef struct {
   Int8    hCode;
   UInt8   hRes;
   Int8    vCode;
   UInt8   vRes;
} vldMVector_t;
```

Fields

`hCode`	See **motion_code[r][s][0]** [6.3.17].
`hRes`	See **motion_residual[r][s][0]** [6.3.17].
`vCode`	See **motion_code[r][s][1]** [6.3.17].
`vRes`	See **motion_residual[r][s][1]** [6.3.17].

Description

Provides horizontal and vertical motion vectors.

For more information on the interpretation of the fields, refer to official MPEG standard: the applicable paragraphs in ISO/IEC draft 13818-2 are included in the above comments.

vldMV_t

```
typedef struct {
   vldMVector_t   forw;
   vldMVector_t   back;
} vldMV_t;
```

Fields

`forw`	Forward motionvector [6.3.17].
`back`	Backward motionvector [6.3.17].

Description

Provides forward and backward (horizontal and backward) motion vectors.

For more information on the interpretation of the fields, refer to official MPEG standard: the applicable paragraphs in ISO/IEC draft 13818-2 are included in the above comments.

vldMBHMpeg1_t

```
typedef struct {
   UInt32   word[4];
} vldMBHMpeg1_t;
```

Fields

`word` — Decoded MPEG-1 macroblock header. The interpretation of the bits in these four words can be found in the appropriate TriMedia data book, Figure 14-2. These four words exclude the MPEG-2-specific second forward and backward motion vectors (words three and five) in Figure 14-2.

Description

Macroblock header produced by the VLD in the output buffer, for an MPEG-1 stream. This raw output can be converted to a **vldMBHField** using the function **vldGetMBHeader**.

vldMBHMpeg2_t

```
typedef struct {
   UInt32   word[6];
} vldMBHMpeg2_t;
```

Fields

`word[6]` — Number of words that each header takes. Note that these six words include the second forward and backward motion vectors (words three and five) in the TriMedia data book, Figure 14-2.

Description

Macroblock header produced by the VLD in the output buffer, for an MPEG 2 stream. This raw output can be converted to a **vldMBHField** using the function **vldGetMBHeader**.

vldMBH_Field_t

```
typedef struct {
   UInt8     mbEscapeCnt;
   UInt8     mbAddrIncr;
   UInt8     mbType;
   UInt8     motionType;
   UInt8     dctType;
   UInt8     mvCount;
   UInt8     mvFormat;
   UInt8     dmvFlag;
   UInt8     quantScaleCode;
   UInt8     cbp;
   Int       dmvector[2];
   UInt8     mvFieldSel[2][2];
   vldMV_t   mv[2];
} vldMBH_Field_t;
```

Field

`mbEscapeCnt`	Number of **macroblock_escape**s [6.3.17.1].
`mbAddrIncr`	See **macroblock_address_increment** [6.3.17.1].
`mbType`	Five bits for **macroblock_quant**, **macroblock_motion_forward**, **macroblock_motion_backward**, **macroblock_pattern**, and **macroblock_intra.** See [6.3.17.1].
`motionType`	See **frame_motion_type** [6.3.17.1].
`dctType`	See **dct_type** [6.3.17.1].
`mvCount`	**motion_vector_count** minus 1. See [6.3.17.2].
`mvFormat`	See **motion_vector_format** [6.3.17.2].
`dmvFlag`	See **dmv** [6.3.17.2].
`quantScaleCode`	See **quantiser_scale_code** [6.3.16].
`cbp`	See **coded_block_pattern** [6.3.17.4].
`dmvector`	See **dmvector** [6.3.17.3].
`mvFieldSel`	See **motion_vertical_field_select** [6.3.17.2].
`mv`	**motion_code** and **motion_residual** r = index [6.3.17.3]

Description

Decoded macroblock header. An instance of this type is the result of transforming the **vldMBHMpeg1_t** or **vldMBHMpeg2_t** using the function **vldGetMBHeader**. For more information on the interpretation of the fields, refer to official MPEG standard: the applicable paragraphs in ISO/IEC draft 13818-2 are included in the above comments.

vldInstanceSetup_t

```
typedef struct {
   intPriority_t   priority;
   UInt32          interrupts;
   pFnVldISR_t     vldISR;
   pFnVldEmpty_t   vldEmptyFunc;
} vldInstanceSetup_t;
```

Fields

`priority`	Defines the priority of the interrupt that is raised if the VLD stops executing.
`interrupts`	Determines upon which conditions the VLD will raise an interrupt. The value can be an OR'd combination of **VLD_IMASK**_*xxx*, where *xxx* is any of `SUCCESS, ERROR, STARTCODE, DMA_IN_DONE, MBH_OUT_DONE, RL_OUT_DONE.`
`vldISR`	Pointer to the interrupt service routine that will be called when the VLD raises an interrupt.
`vldEmptyFunc`	Pointer to a function that is called when the library needs input to execute a function.

Description

An instance of this type is passed to the **vldInstanceSetup** function.

vldContext_t

```
typedef struct {
    UInt32   inputAddr;
    UInt32   inputCnt;
    UInt32   control;
    UInt32   imask;
    UInt32   offset;
    UInt32   fifoLength;
    UInt8   *fifo;
    UInt8    fifoBuf[192];
} vldContext_t;
```

Fields

`inputAddr`	Value of VLD_BIT_ADR register.
`inputCnt`	Count value of VLD_BIT_CNT register.
`control`	Content of the VLD_CTL register.
`imask`	Content of the VLD_IMASK register.
`offset`	Offset from a byte boundary to the start of unused portion bitstream.
`fifoLength`	Number of unused bytes in the shift register VLD_SR.
`fifo`	A pointer to a 64-byte aligned address of buffer containing the unused fifo data (**fifoBuf**).
`fifoBuf`	An array large enough to store 66 bytes of data from an aligned address in the input buffer.

Description

The data stored in this structure must be written at a point where it is safe to perform the context switch. The VLD context consists of the contents of several VLD registers (VLD_BIT_ADR, VLD_BIT_CNT, VLD_CTL, VLD_IMASK) and the unused data portion of the VLD input FIFO content up to a 64-byte boundary. The information of the saved input FIFO data consists of the bit offset from a byte boundary, the byte number from the current byte to a 64-byte boundary and the actual buffer data.

vldInstanceInfo_t

```
typedef struct {
   Bool               used;
   Bool               mpeg2Mode;
   vldPictureInfo_t   vldPictInfo;
   vldInstanceSetup_t vldSetup;
   vldContext_t       vldContext;
} vldInstanceInfo_t;
```

Fields

`used`	Instance usage flag; 1 used, 0 unused.
`mpeg2Mode`	MPEG flag: 1 for MPEG-2, 0 for MPEG-1.
`vldPictInfo`	The content of the VLD picture Info register for an instance.
`vldSetup`	The setup data for an instance.
`vldContext`	The context data for an instance.

Description

This instance information structure represents all the necessary information for each VLD instance (each input bitstream).

VLD API Functions

This section presents the VLD API device library functions.

vldGetCapabilities

```
tmLibdevErr_t vldGetCapabilities(
   pvldCapabilities_t   *cap
);
```

Parameters

`cap`	Pointer to a variable in which to return a pointer to the cabililities data.

Return Codes

`TMLIBDEV_OK`	Success (always returned).

Description

Sets the provided pointer to global capabilities.

vldOpen

```
tmLibdevErr_t vldOpen(
   Int   *instance
);
```

Parameters

instance	Instance pointer.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NO_MORE_INSTANCES	No more instances can be opened; the maximum number of users has been reached already.

Description

Opens an instance of the VLD device. It checks if there is not another opened instance after disabling the interrupts (intClearIEN). Then it restores the interrupt IEN flag (intRestoreIEN).

vldClose

```
tmLibdevErr_t vldClose(
   Int   instance
);
```

Parameters

instance	Device Library instance.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NOT_OWNER	Asserts, in the debug version, if an incorrect instance is passed.

Description

This function shuts down the device and deinstalls the interrupts. It deinstalls the interrupt handler, and closes the intVLD interrupt, if it was opened.

vldInstanceSetup

```
tmLibdevErr_t vldInstanceSetup(
   Int                  Instance,
   vldInstanceSetup_t   *vldsetup
);
```

Parameters

`Instance`	Owner instance.
`vldsetup`	Pointer to **vldInstanceSetup_t** structure.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Asserts, in the debug version, if an incorrect instance is passed.
`VLD_ERR_INIT_REQUIRED`	Asserts, in the debug version, if no **vldOpen** is called.

Description

This function validates the owner, initializes the registers, resets the VLD, opens an interrupt intVLD, and installs the interrupt handler.

vldCommand

```
tmLibdevErr_t vldCommand(
   Int     instance,
   Int32   command
);
```

Parameters

`instance`	Owner instance.
`command`	Word which is given to **VLD_COMMAND** register: **VLD_COMMAND_xxx count_for_command**.

Return Codes

`TMLIBDEV_OK`	Success.
`VLD_ERR_STATUS_ERROR`	Returned when the VLD reported a syntax error in the bitstream.
`VLD_ERR_UNEXPECTED_START_CODE`	Returned when the VLD reported an unexpected startcode in the bitstream.
`VLD_ERR_PREV_COMMAND_NOT_DONE`	Will assert in the debug version when it is called while the VLD is still executing a command.

Description

This function issues a VLD command by calling the **vldSetCOMMAND** macro. The command **VLD_CMD_PARSE is** asynchronous with the function returning immediately. All other commands are synchronous and return upon completed (the status checking of the **vldCommand** is performed using the **vldGetSTATUS** macro).

vldInput

```
tmLibdevErr_t vldInput(
   Int       instance,
   Pointer   readaddr,
   UInt32    readcount
);
```

Parameters

`instance`	Owner instance.
`readaddr`	Input read address.
`readcount`	Input read count.

Return Codes

`TMLIBDEV_OK`	Success.
`VLD_ERR_BIT_CNT_OVERFLOW`	Asserts, in the debug version, when it is called with a readcount value of more than 4095.

Description

Updates the VLD data input address and count. Supplies the VLD with more data by providing the address and size of the full input buffer using the **vldSetBIT_ADR** and **vldSetBIT_CNT** macros. This should normally be called only when the count is zero. Readcount should be 12 bits.

vldReset

```
tmLibdevErr_t vldReset(
   Int   instance,
);
```

Parameters

`instance`	Owner instance.

Return Codes

`TMLIBDEV_OK`	Success.
`VLD_ERR_RESET_FAIL`	Returned if the hardware peripheral did not respond.

Description

This function resets the VLD to its defaults.

Implementation Notes

The data that was in the VLD registers is lost.

vldGetBits

```
UInt vldGetBits(
   Int       instance,
   Int32     numBits,
   UInt32   *bits
);
```

Parameters

`instance`	Owner instance
`numBits`	Number of bits to parse off.
`bits`	Pointer to the number of parsed bits.

Return Codes

`TMLIBDEV_OK`	Success.
`VLD_ERR_NUM_BITS_OVERFLOW`	Will assert in the debug version if it is called with a numBits value of more than 32.

Description

This function parses off the input bitstream the specified number of bits. This function uses the vldCommand function with VLD_COMMAND_SHIFT parameter. A maximum of 32 bits can be parsed at one function call.

Implementation Notes

All of the parsed bits are lost.

vldShowBits

```
UInt vldShowBits(
   Int       instance,
   Int32     numBits,
   UInt32   *bits
);
```

Parameters

`instance`	Owner instance
`numBits`	The number of bits to be shown.
`bits`	Pointer to the value of parsed bits.

Return Codes

`TMLIBDEV_OK`	Success.
`VLD_ERR_SR_OVERFLOW`	Will assert in the debug version if it is called with a numBits value of more than 16.

Description

Reads the bits in the VLD Shift Register with a 16 bit limit and does not shift any bit in the VLD. The function uses the **vldGetSR_VALUE** macro to get the content of the VLD Shift Register.

vldFlushBits

```
tmLibdevErr_t vldFlushBits(
   Int     instance,
   Int32   numBits
);
```

Parameters

`instance`	Owner instance
`numBits`	The number of bits to be flushed.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function parses off and discards **numBits** bits. It uses the vldCommand function with the VLD_COMMAND_SHIFT as many times as required.

vldNextStartCode

```
UInt vldNextStartCode(
   Int       instance,
   UInt32   *startcode
);
```

Parameters

`instance`	Owner instance
`startcode`	Pointer to the startcode value.

Return Codes

`TMLIBDEV_OK`	Success.

Description

Searches for the next **startcode** using the vldCommand function with the VLD_COMMAND_STARTCODE parameter, and combine the next eight bits after the MPEG start code prefix (0x000001) using the **vldShowBits** function. It parses and discard all of the bits until the start code is hit.

vldSetPictureInfo

```
UInt vldSetPictureInfo(
   Int                instance,
   vldPictureInfo_t   *pictInfo
);
```

Parameters

instance	Owner instance.
pictInfo	**vldPictureInfo_t** structure instance.

Return Codes

TMLIBDEV_OK	Success.

Description

This function sets the picture information parameters for the VLD picture information register with the **vldSetPI** macro.

vldGetPictureInfo

```
UInt vldGetPictureInfo(
   Int                instance,
   vldPictureInfo_t  *pictInfo
);
```

Parameters

`instance`	Owner instance
`pictInfo`	**vldPictureInfo_t** structure instance.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function gets the picture information parameters from the VLD picture information register with the **vldExtractPI**_*xxx* macros (refer to tmVLDmmio.h).

vldFlushOutput

```
tmLibdevErr_t vldFlushOutput(
   Int   instance
);
```

Parameters

`instance`	Owner instance

Return Codes

`TMLIBDEV_OK`	Success.

Description

Flushes the output FIFO's data to SDRAM. This function calls vldCommand with VLD_COMMAND_WR_FIFO_FLSH (refer to tmVLD.h).

vldParseMacroblocks

```
tmLibdevErr_t vldParseMacroblocks(
   Int       instance,
   Int32     count,
   Pointer   mbhAddr,
   Pointer   rltokenAddr,
   Int32     mbhBufSize,
   Int32     rlBufSize
);
```

Parameters

`instance`	Owner instance
`count`	The number of macro blocks.
`mbhAddr`	The beginning address of parsed macro block headers.
`rltokenAddr`	The output address of parsed coefficients.
`mbhBufSize`	The size of mbh buffer starting from **mbhAddr** in 32-bit words, 9 bits corresponding. **VLD_STATUS** register bit is set when this buffer is full.
`rlBufSize`	The size of rl buffer (starting from **rltokenAddr**) in 32-bit words, 12 bits corresponding. **VLD_STATUS** register bit is set when this buffer is full.

Return Codes

`TMLIBDEV_OK`	Success.
`VLD_ERR_MBH_CNT_OVERFLOW`	Will assert in the debug version if it is called with a mbhBufSize value of more than 511.
`VLD_ERR_RL_CNT_OVERFLOW`	Will assert in the debug version if it is called with a rlBufSize value of more than 4095.

Description

Parses the **count** number of MacroBlocks. It sets the MMIO registers with the given parameters (**vldSetMBH_ADR**, **vldSetMBH_CNT**, **vldSetRL_ADR**, **vldSetRL_CNT** macros), and sends a VLD_COMMAND_PARSE message via the **vldCommand** function.

vldGetMBHeader

```
tmLibdevErr_t vldGetMBHeader(
   Int              instance,
   vldMBH_Field_t  *mbhField,
   Pointer          mbhAddr
);
```

Parameters

instance	Owner instance.
`mbhField`	Pointer to **vldMBHField_t** structure.
`mbhAddr`	The beginning address of parsed macro block headers.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Can assert if the instance is invalid.

Description

This function gets the macro block header parameters from the macro block header output buffer.

vldSaveContext

```
tmLibdecErr_t vldSaveContext (
   Int   instance,
   Int   offset
);
```

Parameters

`instance`	Owner instance.
`offset`	The bit offset from a byte boundary in the VLD input buffer.

Return Codes

`TMLIBDEV_OK`	Success.
`VLD_ERR_WRONG_OFFSET`	Asserts, in the debug version, if the function is called with the offset value more than 8.
`VLD_ERR_SAVING_CONTEXT_ERROR`	The function failed.

Description

The function saves the context for a specific instance with a given bit offset from a byte boundary in the current VLD input buffer.

vldRestoreContext

```
tmLibdevErr_t vldRestoreContext (
   Int   instance
);
```

Parameters

`instance`	Owner instance.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function restores the context for a specific instance for continuous decoding of associated bitstream.

Chapter 7

Video Transformer (VtransICP) API

Video Transformer API Overview

The TriMedia Video Transformer application library simplifies the filtering and display of video images. The Video Transformer supports output to either SDRAM or ICP-based DMA over the PCI interface. It can send output images to the PC screen over the PCI bus; in this scenario it will perform YUV to RGB color conversion. It can filter YUV images to SDRAM; in this case, the output format is usually YUV.

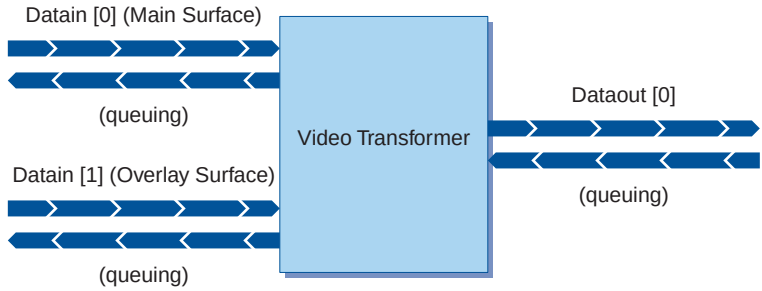


Figure 2 Structure of the Video Transformer

The basic concept behind the development of the video transformer is to reduce the process of video transformation to a simple, high-level interface. There are two main phases: the setup phase, during which the characteristics of the video stream are specified, and the frame transformation phase, which performs filtering and color-space conversion operations. The video transformer supports both the Application Library layer and Operating System Layer of the TriMedia Software Architecture.

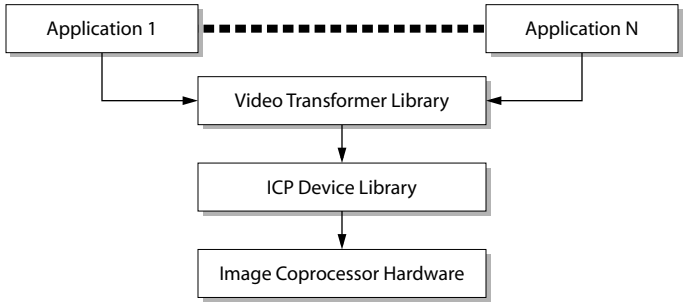


Figure 3 Video Transformer Architecture

Video Transformer Functionality

The TriMedia video transformer provides the following functionality in both the AL (non-streaming) and OL (streaming) layers:

- YUV to YUV vertical scaling.
- YUV to YUV horizontal scaling.
- YUV to RGB horizontal scaling and color conversion, with optional overlay and bit-mask. The output can be to SDRAM or PCI.
- Deinterlace filtering (YUV interlaced to YUV progressive scan).
- YUV anti-flicker filtering (for graphics which are displayed on an interlaced screen).
- YUV422 to YUV420 conversion.
- Accept buffers which do not have strides that are multiples of 64 bytes.
- Scale a YUV image into a subsection of a YUV buffer.
- Copies MPEG-related information from the input packet to the output packet.

Figure 4, a flow diagram, shows video transformer filtering operations.

Limitations

- There are restrictions on the destination PCI address when outputting **vdfRGB24**. The restriction applies when you wish to place the output at an address which is offset from the start of the PCI video buffer. The offset must be a multiple of six bytes (i.e., on an even pixel boundary), otherwise the color will be incorrect. For example, consider a PCI video card which has a base address of 0xE0000000. Valid PCI addresses would be 0xE0000000, 0xE0000006, 0xE000000C, 0xE0000012, etc.
- For MPEG packets, the video transformer copies the data identified by the input packet's **header–>userPointer** to the location identified by the output packet's **header–>userPointer**. The video transformer does not allocate the memory required to store this data in the output packet; the application must allocate this memory and initialize the pointer. If the output packet's **header–>userPointer** is null, the video transformer will not copy the MPEG data.

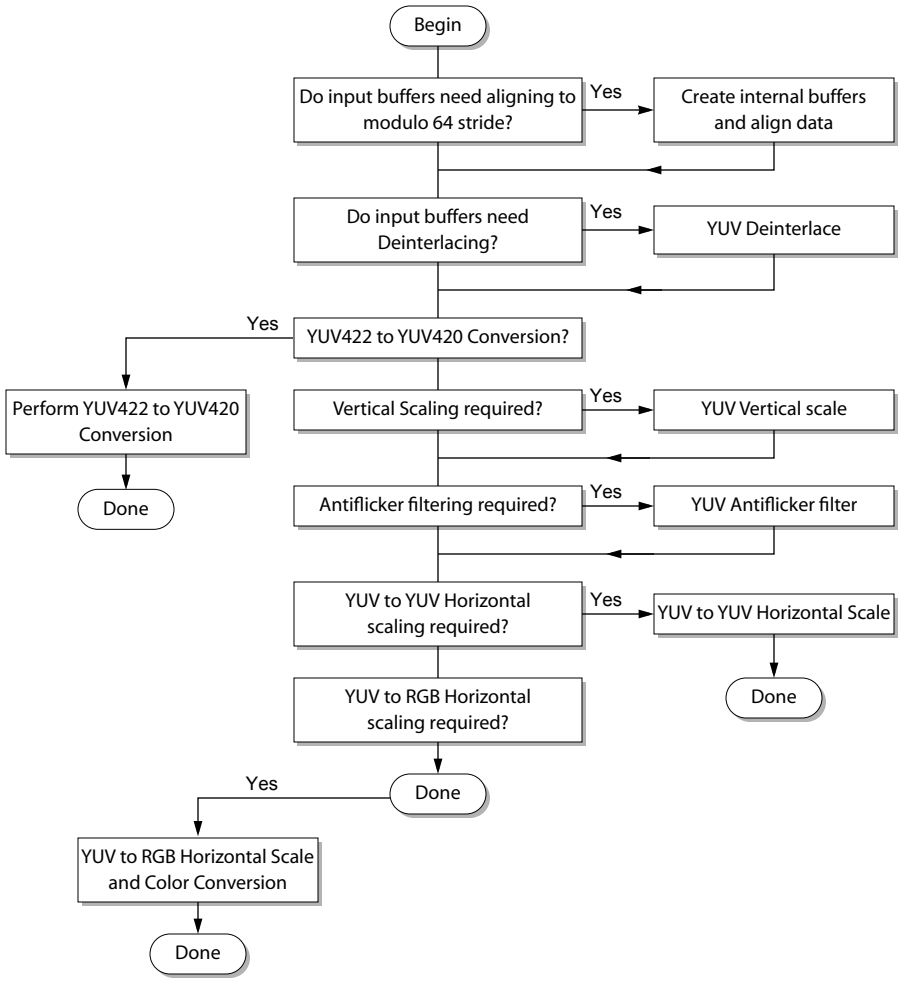


Figure 4 Control flow of video transformer filtering

Using the Video Transformer API

The TriMedia Video Transformer API is contained within the archived application library libtmVtransICP.a. To use the Video Transformer AL layer API, you must include the tmalVtransICP.h header file. For OL layer applications, you must include the tmolVtransICP.h header file.

The AL Layer

The operating system independent layer supports only non-data streaming operation; the application explicitly calls the **tmalVtransICPProcessFrame** function to perform the desired video transformation. A diagram of typical flow of control is shown in Figure 5.

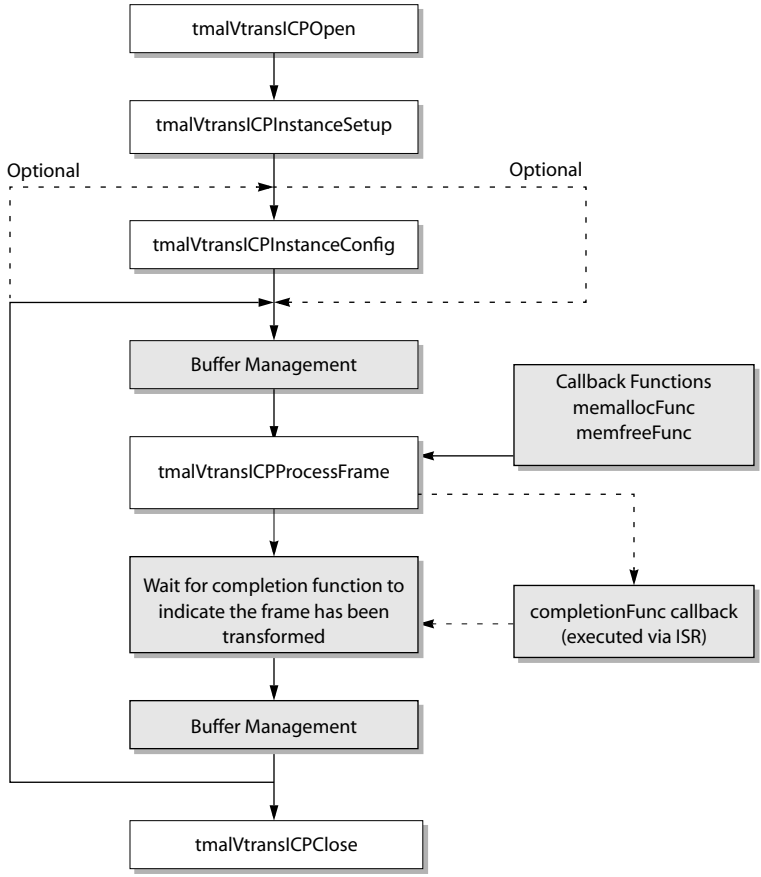


Figure 5 Non-data streaming flow control

First, create an instance of a video transformer by calling the **tmalVtransICPOpen** function; a maximum of four instances can exist. Once an instance has been opened, set it up by calling the function **tmalVtransICPInstanceSetup**. The **tmalVtransICPInstance-Setup_t** structure passed to this function defines the instance context. One of the parameters in this structure is a completion function; since the transformation of a frame is asynchronous, this callback function is used to notify the sender when the frame has been completed.

If the final output of the transformation is to PCI, then the destination address for the transformation must be defined using the **outputPCIAddr** field and a null pointer passed for the **tmalVtransICPProcessFrame** output packet parameter. For output to SDRAM an output packet should be used to specify the destination address and format information; in this case the **outputPCIAddr** field must be set to null.

Note
The input, overlay and output packet structures are defined in tmAvFormats.h.

The **tmalVtransICPInstanceConfig** function can be called to change the configuration of the video transformer once the instance has been setup. For example, it could be used to enable/disable the overlay, or change the overlay position.

The application can then call the **tmalVtransICPProcessFrame** function to initiate the frame transformation. The **progressFunc** callback will be executed once the request has been placed on the ICP queue. The transformation operation is asynchronous to the DSPCPU as it is performed by the ICP coprocessor; the application is informed of completion via the **completionFunc** callback. This function will usually set a flag to indicate processing has finished with the application polling the flag to determine the status of the transformation. Once the transformation has completed, the application may then perform further buffer management, for example, passing the packet on for further processing by another component. The application can repeatedly transform frames by simply calling the **tmalVtransICPProcessFrame** function with the relevant packet parameters. It may also alter the image and overlay parameters by calling **tmalVtransICPInstanceConfig** before processing the frame. Finally, the instance can be destroyed by calling the **tmalVtransICPClose** function.

The OL Layer

The OL layer supports data streaming operation with message queues being used to transfer packets of data between components. The tmolDefaults library is used to provide default callback functions; this means that the application programmer simply has to create the In/Out Descriptor and connect the relevant components to it. Figure 6 on page 79 shows typical function call control flow.

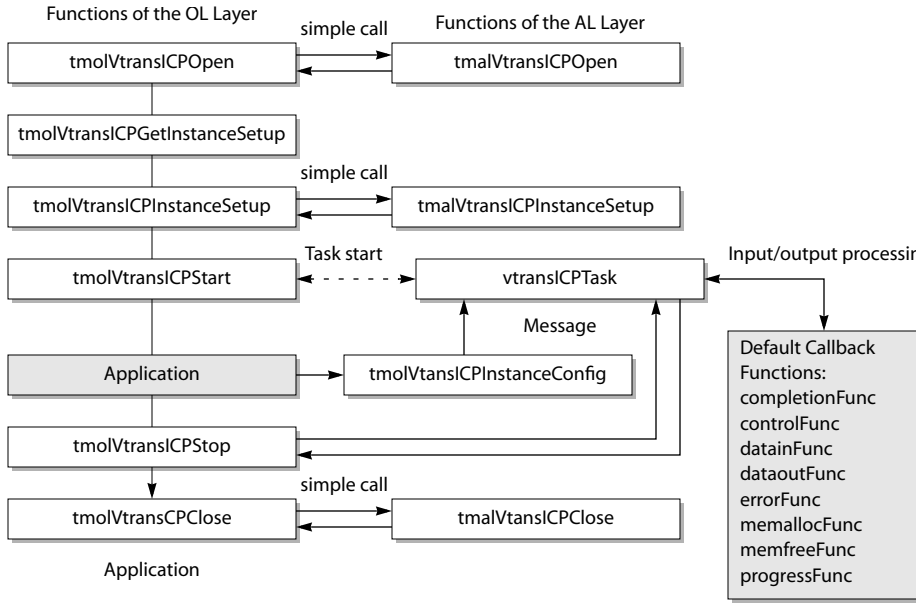


Figure 6 OL Layer Flow Control

The application first opens an instance of the Video Transformer using **tmolVtransICPOpen**. The **tmolVtransICPGetInstanceSetup** function should then be called. This returns a pointer to the instance variables, which were created when the instance was opened. The application should initialize instance variable fields required for setup of the instance. For example, the **outputPCIAddr** field must be initialized if the final output is to a PCI video card. The **tmolVtransICPInstanceSetup** function is then called to initialize the instance variables.

Data streaming is initiated by calling **tmolVtransICPStart**; this creates a separate operating system task which executes the Video Transformer code and runs in parallel with the application task. The application is then free to perform independent processing, while the Video Transformer is streaming data; it may call the **tmolVtransICPInstanceConfig** function to modify the behavior of the instance. For example, it could change the alpha values used for the overlay.

The Video Transformer can be stopped by calling **tmolVtransICPStop**; this will send a stop request to the Video Transformer task via an operating system message. The Video Transformer will send an acknowledge back to the application indicating that data streaming has stopped. Finally, the instance can be closed by calling **tmolVtransICPClose**.

Callback Function Requirements

This section describes the callback function sequence during data streaming operation. This information is provided to enable the application programmer to understand when

the callback functions are used. A flow diagram of the streaming task operation is shown in .

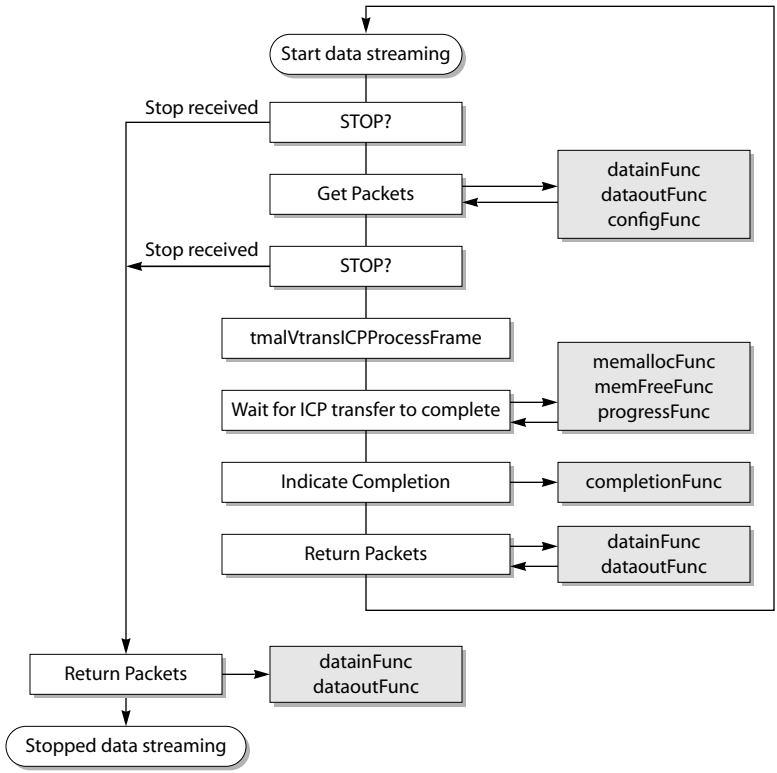


Figure 7 Flow diagram of data streaming. Grey boxes represent callback functions.

The initial data packets are obtained using the datain and dataout callback functions. The **datainFunc** is always called to obtain the main image input packet. The optional overlay and output packets are also obtained via the **datainFunc** and **dataoutFunc** functions respectively. The default callback functions detect if a stop request has been sent; if it has then no further packets are obtained and the data streaming stop operation is performed. The **tsaDatainCheckControl** flag is set when the **datainFunc** function is called; this indicates to the default callback function that it should check to see if a configuration command has been sent to the video transformer instance. The OL layer of the video transformer uses command queues to pass configuration information to the instance. The **configFunc** is automatically called when a command is detected; this is mapped directly to the **tmalVtransICPInstanceConfig** function.

Once the required packets have been obtained, the **tmalVtransICPProcessFrame** function will be executed. This uses **memallocFunc** and **memfreeFunc** to allocate any internal buffers which are required. The **progressFunc** will be called once the video transformation request has been placed on the ICP queue.

The **completionFunc** is executed once the ICP request has been serviced. The **datainFunc** and **dataoutFunc** callbacks are then used to place the full and empty packets on the respective queues. The default functions will check to see if a data streaming stop request has been made; if it has the streaming loop will be exited.

Once a stop request is detected, any packets that are being held by the video transformer are returned using the **datainFunc** and **dataoutFunc** functions.

Packet Formats

The Video Transformer uses the standard packet data types defined in the tmAvFormats.h include file. Both YUV and RGB data use the **tmAvPacket_t** structure; RGB packets have only a single buffer to store the RGB data. YUV data is stored in three buffers, with the Y pointer contained in buffer[0] and the UV pointers contained in buffer[1] and buffer[2] respectively.

Each packet contains a header structure providing information concerning the packet data. An important field in this header is the format field. For the Video Transformer, this is a pointer to a **tmVideoFormat_t** structure which provides information about the video format and image size. There are restrictions on the type of video formats for the Video Transformer inputs and output; these are described below.

Main Image Input Packet

The main image input packet must be either YUV422 or YUV420. The packet headers format field should be pointer to a **tmVideoFormat_t** structure and initialized with the following values:

Field	Value
dataClass	avdcVideo
dataType	vtfYUV
dataSubtype	vdfYUV422Planer or vdfYUV420Planer
description	Null or vdfInterlaced
imageWidth	Width of video frame (luminance)
imageHeight	Height of video frame (luminance)
imageStride	Stride of video frame (luminance)
imageUVstride	Stride of video frame (chrominance)

Overlay Input Packet

The overlay input packet must be either YUV or RGB depending on the output format. The rule is that if the output is YUV then the overlay must be YUV; similarly, if the output is RGB then the overlay must be RGB. This is shown in the table below:

Output Format	Overlay Format
vdfRGB8A_233	vdfRGB15Alpha or vdfRGB24Alpha
vdfRGB8R_332	
vdfRGB15Alpha	
vdfRGB16	
vdfRGB24	
vdfRGB24Alpha	
vdfYUV422Sequence	vdfYUV422SequenceAlpha
vdfYUV422SequenceAlpha	

The packet header's format field should be pointed to a **tmVideoFormat_t** structure and initialized with the values in the table below:

Field	Value
dataClass	avdcVideo
dataType	vtfYUV or vtfRGB
dataSubtype	vdfYUV422SequenceAlpha,
	vdfRGB15Alpha
	vdfRGB24Alpha
description	NULL
imageWidth	Width of overlay frame
imageHeight	Height of overlay frame
imageStride	Stride of overlay frame

Output Packet

The output packet is only used when SDRAM is the destination of the transformation; the data type may be either YUV or RGB. The packet header's format field is automati-

cally placed on a full packet by the default dataout callback function. The format points to a **tmVideoFormat_t** structure and will be initialized with the following values:

Field	Value
dataClass	avdcVideo
dataType	vtfYUV or vtfRGB
dataSubtype	vdfYUV422Sequence
	vdfYUV422SequenceAlpha
	vdfRGB8A_233
	vdfRGB8R_332
	vdfRGB15Alpha
	vdfRGB24
	vdfRGB24Alpha
	vdfYUV422Planer
	vdfYUV420Planer
	vdfRGB16
description	NULL
imageWidth	Width of output frame
imageHeight	Height of output frame
imageStride	Stride of video frame
activeVideoStartX	Set to Zero by Video Transformer
activeVideoStartY	Set to Zero by Video Transformer
activeVideoEndX	Set to imageWidth-1 by Video Transformer
activeVideoEndY	Set to imageHeight-1 by Video Transformer
imageUVstride	Stride of the video frame (chrominance) when the output is YUV. Not used when the output is RGB

When using the OL layer and output to SDRAM, the Video Transformer automatically installs the output format on the IO descriptor. This operation is performed during instance setup and also when the output format is changed via the **tmolVtransICP-InstanceConfig** function.

Scaling to a Sub-Section of a YUV Buffer

The Video Transformer is capable of scaling a YUV image to a sub-section of a YUV buffer located in SDRAM. An example of the use of this functionality is where a video stream is scaled to a corner of an output frame and is surrounded by graphics; the Video Trans-

former scales the video and stores it in the desired location of the output buffer. This is shown in Figure 8.

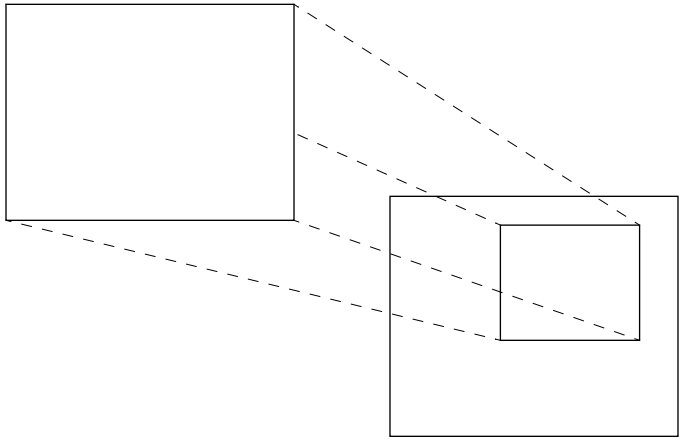


Figure 8 Scaling to a Subsection of a YUV Buffer

The application specifies the size and position of the scaled image by using the **activeVideo** fields in the **tmalVtransICPInstanceSetup_t**/**tmolVtransICPInstanceSetup_t** outputFormat structure. The four parameters are described in Table 1.

Table 1 Sub-Section Scaling Parameters

Field	Description
activeVideoStartX	Left horizontal position
activeVideoStartY	Top vertical position
activeVideoEndX	Right horizontal position
activeVideoEndY	Bottom vertical position

These parameters specify the coordinates of the image in the output buffer. From these, the scaled output width and height are determined. To scale the original image to the entire output buffer, the application should set the **activeVideoStartX**/**activeVideoStartY** parameters to 0 and the **activeVideoEndX**/**activeVideoEndY** to **imageWidth** – 1 and **imageHeight** – 1 respectively. To scale to a subsection, the application should set the fields to the desired positions in the output buffer.

The initial output image size it set during the call to the instance setup function (**tmalVtransICPInstanceSetup**/**tmalVtransICPInstanceSetup**). The application can dynamically change the size and position by using the instance config functions (**tmalVtransICPInstanceConfig**/**tmolVtransICPInstanceConfig**).

The Video Transformer will set the area of the output buffer which is not occupied by the scaled image to black (Y = 0x10, U/V = 0x80). For performance reasons, this “background

filling" is only applied to the output packet when the output packet has not been previously filled or when the location/size of the scaled video changes.

The sub-scaling functionality only applies to YUV scaling to SDRAM. It is not required for output to PCI as the application can simply use the **imageWidth**, **imageHeight**, **imageStride** and **outputPCIAddr** parameters to perform the desired scaling into the PCI video card frame buffer.

Buffer Alignment, Stride, and Cache Coherency

The Video Transformer expects all pointers to the YUV and RGB frame components to be aligned on 64 byte boundaries. The application can ensure this by using the **_cache_malloc** and **_cache_free** functions provided with the software development environment.

The video transformer does not place any restrictions on the memory stride of the YUV and RGB frame components. It will check the stride of the YUV buffers and automatically align them if vertical processing is required (e.g. vertical scale, deinterlace, anti-flicker).

For example, consider a YUV image stored in a buffer with a width of 720 pixels, height 480 lines, and a stride of 720 bytes. If vertical scaling is required, then the YUV image must be stored in a buffer which has a stride with multiples of 64 bytes (in this case, 768 bytes for the Y data and 384 bytes for the U/V data). The video transformer will automatically create internal buffers with the required strides and move the image to these buffers before scaling is performed.

The user should be aware that this automatic alignment has a memory and bandwidth cost associated with it. The memory overhead is the internal buffers which will be created that are used to store the aligned image. Additional bandwidth is required to copy the image from the unaligned buffer to the aligned buffer. If the application provides buffers with strides which have the correct modulo 64 alignment then there is no memory or bandwidth overhead.

When the application programmer is using only the AL-layer, while using the CPU to manipulate the video data before video transformation, the application must flush the buffers out of the CPU data cache before any ICP requests are made. This is because there is no cache coherency between the CPU data cache and the ICP. This coherency can easily be maintained by using the **_cache_copyback** function once the application has completed it's manipulation of the video buffers. Similarly, if the application uses the DSPCPU to manipulate the video data after processing by the video transformer, the application must perform a **_cache_invalidate** on the buffer before processing by the DSPCPU.

When using the OL-layer, cache coherency is maintained automatically between connected components.

Demonstration Programs

The Video Transformer application library contains two example programs, **exalVtransICP** and **exolVtransICP**, which are located in the example tree of the TriMedia Application System.

The source code for the example programs is provided with the library. The programs demonstrate the use of various Video Transformer APIs, including non-data streaming and data streaming operation, and serve as an example for developers who want to use the Video Transformer in their applications.

AL Layer Example

The AL layer example can be found in the examples/exalVtransICP directory, and is called exalVtransICP.c. The example reads two image files from disk, converts the overlay image from YUV to RGB format, then combines the two images and puts the result on the PC screen. The main image is scaled to various sizes, while the overlay image position is moved.

Running the Example

The example can be executed using either tmgmon or tmrun. The command line arguments are as follows:

```
exalVtransICP.out [-help] -d <address> -s <stride> -mode <pci_mode>
[-path <path>]
```

The **–help** option will print out the program arguments and terminate.

The **–d** *address* argument specifies the address of the PCI video card. The argument is mandatory. The address should be specified in hexadecimal, e.g., –d 0xFE000000.

The **–s** *stride* argument specifies the stride of the PCI video card. This is dependent on the resolution of the display and the number of bytes per pixel. e.g., with a screen resolution of 1024 pixels and two bytes per pixel (16-bit color) the stride would be set to **–s 2048**. The argument is mandatory.

The **–mode** *pci_mode* specifies the screen mode of the PCI video card and is mandatory. The *pci_mode* is an integer from 1–4 and represents the following modes:

- RGB 24+alpha
- RGB 24
- RGB 15+alpha
- RGB 16

The **–path** *path* allows the user to specify an alternate path to where the required data files are stored. The program requires the clown640.y, clown640.u, clown640.v,

clown120x100.y, clown120x100.u, and clown120x100.v data files. By default it will search in the current directory and in the data/video directory of the application tree.

exalVtransICP Program Flow

The AL layer control flow is shown in the following diagram:

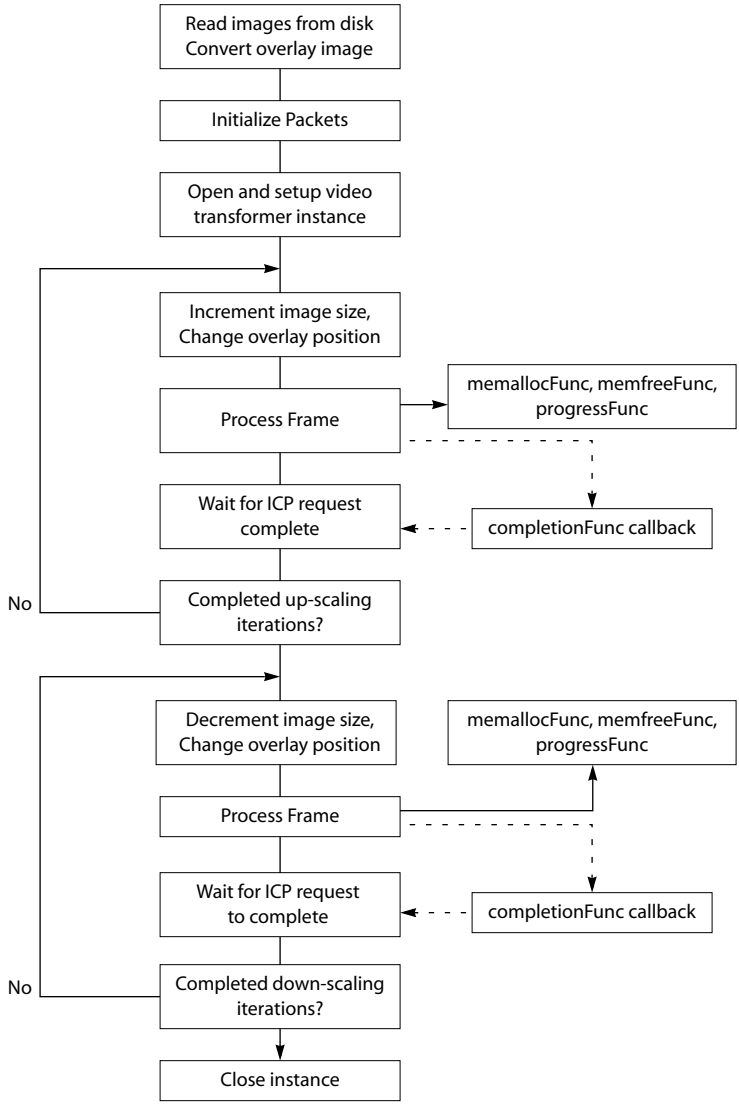


Figure 9 AL layer non-data streaming example program.

The two images used for the main display and overlay are read into memory and the overlay image converted from YUV to RGB format. This is necessary as the output of the video transformer is to the PCI bus which requires RGB, and hence, the overlay must be of the same type.

The packets used for the YUV and overlay data are then created and initialized.

An instance of a video transformer is opened and initialized; this consists of setting up the instance variables such as the callback function pointers. The code then enters a loop where the main image is scaled up over a number of iterations.

The first operation inside the loop is to alter the main image output size and the position of the overlay. This is performed using the **tmalVtransICPInstanceConfig** function to change the instance parameters. The frame is then processed using the **tmalVtransICP-ProcessFrame** function and the program will then wait for the completion function to indicate that the request has been completed.

Note
The application must provide memallocFunc, memfreeFunc, errorFunc, progressFunc, and completionFunc callback functions.

Once the required number of up-scaling operations have completed, the process is reversed with down scaling being performed using an identical sequence of operations. Finally, the instance is closed once the required number of iterations has completed.

OL Layer Example

The OL layer example can be found in the examples/exolVtransICP directory, and is called exolVtransICP.c. A block diagram of the main tasks is shown in Figure 10:

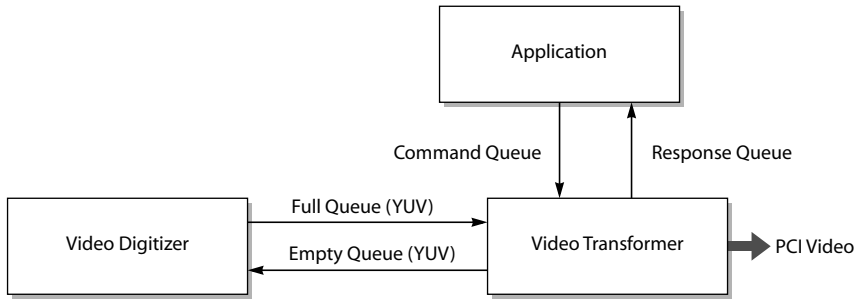


Figure 10 Message queues between library components.

The example uses the video digitizer and the video transformer to generate live video on the PC screen. The video digitizer acts as a source, producing video frame packets obtained through the video input hardware; these packets are placed on its full queue.

The video transformer will obtain packets generated by the digitizer on its main input. It will then scale the main image and then perform RGB color conversion. Its output will be over the PCI bus.

The packets generated by the video digitizer contain interlaced video. By default, the video transformer simply scales the images and displays them on the PC, ignoring the fact that it is displaying interlaced video as progressive video. This will cause movement artifacts to be seen. To overcome this, the user can type commands on the keyboard which toggle the use of the deinterlace functionality contained within the video transformer library. When enabled, the interlaced video is converted to non-interlaced (progressive) before scaling and RGB conversion. The user can also toggle the anti-flicker filter functionality.

Running the Example

The example can be executed using either tmgmon or tmrun. The command line arguments are:

```
exolVtransICP.out [-help] -d <address> -s <stride> -mode <pci_mode>
```

The **–help** option will print out the program arguments and terminate.

The **–d** *address* argument specifies the address of the PCI video card. The argument is mandatory. The address should be specified in hexadecimal, e.g. **–d 0xfe000000**.

The **–s** *stride* argument specifies the stride of the PCI video card. This is dependent on the resolution of the display and the number of bytes per pixel. e.g. with a screen resolution of 1024 pixels and two bytes per pixel (16 bit color) the stride would be set to **–s 2048**. The argument is mandatory.

The **–mode** *pci_mode* specifies the screen mode of the PCI video card and is mandatory. The *pci_mode* is an integer from 1–4 and represents the following modes:

- RGB 24+alpha
- RGB 24
- RGB 15+alpha
- RGB 16

When run, the user can type commands through the TriMedia Console window. The following commands are available:

A	Toggle antiflicker filter
D	Toggle deinterlace filter
I	Disable both antiflicker and deinterlace filtering (display interlaced)
Q	Quit

The commands are case-insensitive and should be followed by pressing the return key.

exolVtransICP Program Flow

A diagram showing the program flow is shown below in figure Figure 11.

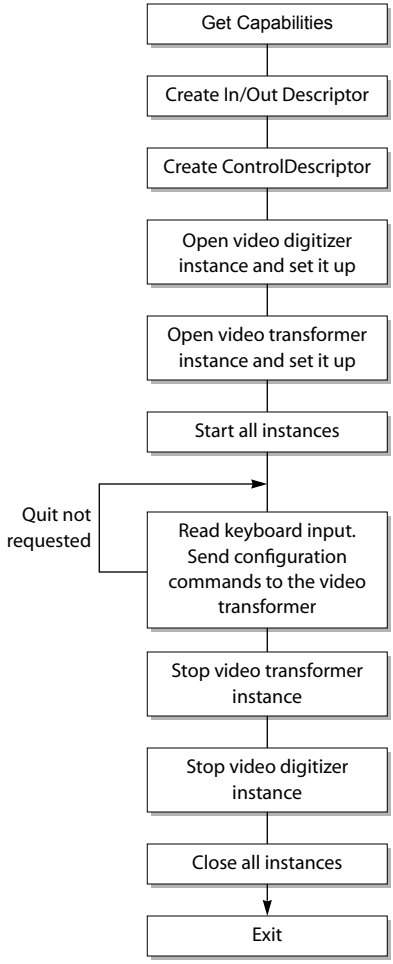


Figure 11 OL layer example program.

The program first obtains the capabilities of the two components which it will connect. It then creates the In/Out Descriptor which is used to describe the connection between the two components; this will also automatically create the packets and place them in the empty queue. The ControlDescriptor for the video transformer instance is then created; this is used to send control messages from the application to the video transformer. As the video transformer output is to PCI, no further In/Out Descriptors are necessary.

Next, an instance of the video digitizer is opened, its instance variable pointer is obtained, and the instance is initialized. The instance setup simply involves initializing the In/Out Descriptor.

The video transformer instance is opened, and a pointer to its instance variables is obtained. The input descriptor is initialized in a similar manner to the video digitizer.

Note
This instance has two inputs: one for the main image, and one for the overlay. In this example, the main image is active while the overlay image is unused.

As this instance is writing its output to the PCI bus, no output In/Out Descriptor is used. Therefore, it is necessary to initialize the output parameters in the instance setup.

All the instances are now ready to begin data streaming, and each instance is explicitly started. The application will wait for the user to type a command through the keyboard. The antiflicker, deinterlace, and interlace commands will cause the application to call the **tmolVtransICPInstanceConfig** function. This will send the requested command to the video transformer instance.

Data streaming will continue until the user presses the 'Q' key on the PC keyboard. When this occurs, the application will simply call the stop functions for the video transformer and video digitizer respectively.

Finally, all the instances are closed, and the In/Out Descriptor and ControlDescriptor are destroyed.

Video Transformer API Data Structures

This section describes the Video Transformer application layer data structures. These data structures are defined in the tmalVtransICP.h header file.

tmalVtransICPOutputType_t

```
typedef enum {
   tmalVtransICPSDRAM  = icpSDRAM,
   tmalVtransICPPCI    = icpPCI
} tmalVtransICPOutputType_t;
```

Description

This enum, used by the struct **tmalVtransICPInstanceSetup_t**, specifies whether the output is being directed to the SDRAM or the PCI bus.

tmalVtransICPOverlayPosition_t

```
typedef struct {
   Int   startX;
   Int   startY;
} tmalVtransICPOverlayPosition_t;
```

Fields

`startX`	The horizontal start position of the overlay.
`startY`	The vertical start position of the overlay.

Description

This structure specifies the top left position of the overlay.

tmalVtransICPAlpha_t

```
typedef struct {
   Float   alpha0;
   Float   alpha1;
} tmalVtransICPAlpha_t;
```

Fields

`alpha0`	The alpha value to use when the alpha bit is 0.
`alpha1`	The alpha value to use when the alpha bit is 1.

Description

This struct specifies the alpha0 and alpha1 values which are used when blending the overlay.

Note
In RGB24+alpha mode, the alpha value is contained in the overlay data.

tmalVtransICPBitMaskSetup_t

```
typedef struct {
   UInt8   *bitMaskBase;
   Int      stride;
} tmalVtransICPBitMaskSetup_t;
```

Fields

`bitMaskBase`	Pointer to the bitmask image.
`stride`	Stride of the bitmask image, usually 1/8 image width.

Description

This struct is used to specify the bitmask address and the bitmask stride.

tmalVtransICPCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
   Int                         granularityOfAddress;
   Int                         granularityOfStride;
} tmalVtransICPCapabilities_t, *ptmalVtransICPCapabilities_t;
```

Fields

`defaultCapabilities`	The default capabilities structure is defined in the Application Layer Library API (tsa.h).
`granularityOfAddress`	Alignment. A value of 6 implies 64-byte alignment (i.e., the low 6 bits of the input address must be zero).
`granularityOfStride`	Restriction on the input stride in terms of byte alignment.

Description

This struct returns the capabilities of the Video Transformer. The default capabilities struct is defined in tsa.h. This struct is used in the function **tmalVtransICPGetCapabilities**.

tmalVtransICPInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t      defaultSetup;
   tmalVtransICPOutputType_t       outputDest;
   UInt8                          *outputPCIAddr;
   Bool                            overlayEnable;
   Bool                            bitMaskEnable;
   Bool                            deinterlaceEnable;
   Bool                            antiflickerEnable;
   Bool                            useDSPCPUDeinterlace;
   Bool                            notConservative;
   tmVideoFormat_t                 outputFormat;
   tmalVtransICPOverlayPosition_t  overlayPosition;
   tmalVtransICPAlpha_t            overlayAlpha;
   tmalVtransICPBitMaskSetup_t     bitMask;
} tmalVtransICPInstanceSetup_t;
```

Fields

`defaultSetup`	Pointer to the struct containing default setup information.
`outputDest`	Specifies the output destination. Either **tmalVtransICPPCI** or **tmalVtransICPSDRAM**.
`outputPCIAddr`	Output address of the PCI video. Must be set to Null if the output is to SDRAM.
`overlayEnable`	Use overlay if TRUE.
`bitMaskEnable`	Use bitmask information if TRUE.
`deinterlaceEnable`	Enable deinterlacing if TRUE.
`antiflickerEnable`	Enable the antiflicker filter if TRUE.
`useDSPCPUDeinterlace`	If TRUE, this specifies that the deinterlacing should be performed by the DSPCPU rather than the ICP. This functionality is currently not supported.
`notConservative`	If TRUE, this specifies that video transformer is allowed to swap YUV packet pointers. This functionality is currently not supported.
`outputFormat`	Specifies the output image format. The supported av subtypes are **vdfRGBA_233**, **vdfRGBR_332**, **vdfRGB15Alpha**, **vdfRGB24**, **vdfRGB24Alpha**, **vdfYUV422Sequence**, **vdfYUV422SequenceAlpha**, **vdfYUV422Planar**, and **vdfYUV420Planar**.
`overlayPosition`	Specifies that top left position of the overlay.

`overlayAlpha`	Specifies the alpha0 and alpha1 values used for the overlays. The values should be between zero and one.
`bitMask`	Specifies the bitmask address and the bitmask stride.

Description

This struct is used in the function **tmalVtransICPInstanceSetup** and specifies the initial configuration of the video transformer instance.

The **deinterlaceEnable** indicates to the video transformer instance that is should deinterlace any packets which have format description field set to **vdfInterlaced**. If this field is not set in the packet format, deinterlacing will not occur.

The **antiflickerEnable** indicates that the video transformer should perform antiflicker filtering on the image. The purpose of this is to reduce the flicker which can occur when computer generated images are displayed on an interlaced screen.

tmalVtransICPConfigTypes_t

```
typedef enum {
   VTRANS_CONFIG_OVERLAY_ENABLE,
   VTRANS_CONFIG_BITMASK_ENABLE,
   VTRANS_CONFIG_DEINTERLACE_ENABLE,
   VTRANS_CONFIG_ANTIFLICKER_ENABLE,
   VTRANS_CONFIG_USE_DSPCPU_DEINTERLACE,
   VTRANS_CONFIG_NOT_CONSERVATIVE,
   VTRANS_CONFIG_INPUT_FORMAT,
   VTRANS_CONFIG_OUTPUT_FORMAT,
   VTRANS_CONFIG_OVERLAY_FORMAT,
   VTRANS_CONFIG_OVERLAY_POSITION,
   VTRANS_CONFIG_OVERLAY_ALPHA,
   VTRANS_CONFIG_BITMASK
} tmalVtransICPConfigTypes_t;
```

Description

This enum is used as a parameter to the **tmalVtransICPInstanceConfig** and **tmolVtransICP-InstanceConfig** functions. The values should be used to specify the **command** field of the **tsaControlArg_t** structure.

tmolVtransICPCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
   Int                         granularityOfAddress;
   Int                         granularityOfStride;
} tmolVtransICPCapabilities_t; *ptmolVtransICPCapabilities_t;
```

Fields

`defaultCapabilities`	The default capabilities structure is defined in the OS Layer Library API (tsa.h).
`granularityOfAddress`	Alignment. A value of 6 implies 64-byte alignment (i.e., the low 6 bits of the input address must be zero).
`granularityOfStride`	Restriction on the input stride in terms of byte alignment.

Description

This structure is used to return the capabilities of the Video Transformer. The default capabilities structure is defined in tsa.h. This structure is used in the function **tmolVtransICPGetCapabilities**.

tmolVtransICPInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t      defaultSetup;
   tmalVtransICPOutputType_t       outputDest;
   UInt8                          *outputPCIAddr;
   Bool                            overlayEnable;
   Bool                            bitMaskEnable;
   Bool                            deinterlaceEnable;
   Bool                            antiflickerEnable;
   Bool                            useDSPCPUDeinterlace;
   Bool                            notConservative;
   tmVideoFormat_t                 outputFormat;
   tmalVtransICPOverlayPosition_t  overlayPosition;
   tmalVtransICPAlpha_t            overlayAlpha;
   tmalVtransICPBitMaskSetup_t     bitMask;
tmolVtransICPInstanceSetup_t, *ptmolVtransICPInstanceSetup_t;
```

Fields

`defaultSetup`	Pointer to the structure containing default setup information.
`outputDest`	Specifies the output destination. Either **tmalVtransICPPCI** or **tmalVtransICPSDRAM**.
`outputPCIAddr`	Output address of the PCI video. Must be set to Null if the output is to SDRAM.
`overlayEnable`	Use overlay if TRUE.
`bitMaskEnable`	Use bitmask information if TRUE.
`deinterlaceEnable`	Enable deinterlacing if TRUE.
`antiflickerEnable`	Enable the antiflicker filter if TRUE.
`useDSPCPUDeinterlace`	If TRUE, this specifies that the deinterlacing should be performed by the DSPCPU rather than the ICP. This functionality is currently not supported.
`notConservative`	If TRUE, this specifies that video transformer is allowed to swap YUV packet pointers. This functionality is currently not supported.
`outputFormat`	Specifies the output image format. The supported av subtypes are **vdfRGBA_233**, **vdfRGBR_332**, **vdfRGB15Alpha**, **vdfRGB24**, **vdfRGB24Alpha**, **vdfYUV422Sequence**, **vdfYUV422SequenceAlpha**, **vdfYUV422Planar**, and **vdfYUV420Planar**.
`overlayPosition`	Specifies that top left position of the overlay.

`overlayAlpha`	Specifies the alpha0 and alpha1 values used for the overlays. The values should be between zero and one.
`bitMask`	Specifies the bitmask address and the bitmask stride.

Description

This structure is used in the function **tmolVtransICPInstanceSetup** to setup the initial configuration of the instance. The application does not need to allocate memory for this structure as it is automatically created during the **tmolVtransICPOpen** function call. The application can obtain a pointer to this structure by calling **tmolVtransICPGetInstanceSetup**.

Note
This structure is identical to the tmalVtransICPInstanceSetup_t structure.

The **deinterlaceEnable** indicates to the video transformer instance that is should deinterlace any packets which have format description field set to **vdfInterlaced**. If this field is not set in the packet format, deinterlacing will not occur.

The **antiflickerEnable** indicates that the video transformer should perform antiflicker filtering on the image. The purpose of this is to reduce the flicker which can occur when computer generated images are displayed on an interlaced screen.

Video Transformer API Functions

This section describes the TriMedia Video Transformer device library API functions.

tmalVtransICPOpen

```
tmLibappErr_t tmalVtransICPOpen(
   Int   *instance
);
```

Parameters

`instance`	Pointer to the instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation failed while creating the instance variables.
`VT_ERR_NO_FREE_INST`	The maximum number of transformers are already allocated.
`VT_ERR_DEVICE_LIBRARY_ERROR`	Some other process is already using the ICP.

Description

This function will assign an instance for usage. A maximum number of **VR_MAX_VIDTRANS** are available. This function will open the ICP with the **icpOpen** device library function if this is the first video transformer instance to be opened.

tmalVtransICPClose

```
tmLibappErr_t tmalVtransICPClose(
   Int   instance
);
```

Parameters

instance	The instance.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	Instance asked to close is not open.
VT_ERR_ICP_REQUEST_IN_PROGRESS	The instance has an ICP request on the queue.
VT_ERR_DEVICE_LIBRARY_ERROR	Some other process is already using the ICP.

Description

This function will deassign instances for later reuse. It will close the ICP device if all instances of the Video Transformer are closed.

tmalVtransICPGetCapabilities

```
tmLibappErr_t tmalVtransICPGetCapabilities(
   ptmalVtransICPCapabilities_t  **tmalVtransICPCap
);
```

Parameters

`tmalVtransICPCap`	Pointer to a variable in which to return a pointer to capabilities data.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Provided so that a system resource controller can obtain information about the video transformer library before installing it. Fills in the address of a static capabilities structure. The **tmalVtransICPCap** pointer is valid until the video transformer library is unloaded.

tmalVtransICPInstanceSetup

```
tmLibappErr_t tmalVtransICPInstanceSetup(
   Int                              instance,
   tmalVtransICPInstanceSetup_t   *setup
);
```

Parameters

instance	Instance value.
setup	Pointer to the setup data.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	Attempt to set up an invalid instance.
TMLIBAPP_ERR_NULL_MEMALLOCFUNC	Memory allocation function pointer not setup.
TMLIBAPP_ERR_NULL_MEMFREEFUNC	Memory free function pointer not setup.
TMLIBAPP_ERR_NULL_ERRORFUNC	Error function pointer not setup.
TMLIBAPP_ERR_NULL_PROGRESSFUNC	Progress function pointer not setup.
VT_ERR_ILLEGAL_STRIDE	Can assert when the output stride is zero.
VT_ERR_ILLEGAL_WIDTH	Can assert when the output width is zero.
VT_ERR_ILLEGAL_HEIGHT	Can assert when the output height is zero.
VT_ERR_ICP_REQUEST_IN_PROGRESS	The instance has an ICP request on the queue.
VT_ERR_DEVICE_LIBRARY_ERROR	Error returned from a call to the ICP device library. Refer to the icpInstanceSetup and icpLoadCoeff functions in the ICP device library documentation.

Description

This function will prepare the Video Transformer for operation by configuring the ICP, loading the default ICP coefficients, and installing the ICP interrupts. The setup structure variables are used to initialize the instance parameters. This function should be called once to setup the instance; further modifications should be made using the **tmalVtransICPInstanceConfig** function.

tmalVtransICPGetInstanceConfig

```
tmLibappErr_t tmalVtransICPGetInstanceConfig(
   Int                instance,
   ptsaControlArgs_t  args
);
```

Parameters

`instance`	Instance value.
`args`	Pointer to a **tsaControlArgs_t** struct specifying the information required.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Attempt to set up an invalid instance.
`TMLIBAPP_ERR_INVALID_COMMAND`	Unrecognized command.

Description

This function will return the current value of specific Video Transformer instance parameters. It uses the standard **tsaControlArgs_t** structure defined in the tsa.h include file to specify the information required. The control structures command field should be set to one of the values specified by **tmalVtransICPConfigTypes_t**. The parameter field should point to a variable where the information will be stored. The following table specifies the command field values with the associated parameter pointer field:

Command	Parameter
VTRANS_CONFIG_OVERLAY_ENABLE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_BITMASK_ENABLE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_DEINTERLACE_ENABLE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_ANTIFLICKER_ENABLE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_USE_DSPCPU_DEINTERLACE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_NOT_CONSERVATIVE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_INPUT_FORMAT	Pointer to tmVideoFormat_t
VTRANS_CONFIG_OUTPUT_FORMAT	Pointer to tmVideoFormat_t
VTRANS_CONFIG_OVERLAY_FORMAT	Pointer to tmVideoFormat_t

Command	Parameter
VTRANS_CONFIG_OVERLAY_POSITION	Pointer to tmalVtransICPOverlayPosition_t
VTRANS_CONFIG_OVERLAY_ALPHA	Pointer to tmalVtransICPAlpha_t
VTRANS_CONFIG_BITMASK	Pointer to tmalVtransICPBitMaskSetup_t

tmalVtransICPInstanceConfig

```
tmLibappErr_t tmalVtransICPInstanceConfig(
   Int                instance,
   ptsaControlArgs_t  args
);
```

Parameters

`instance`	The instance.
`args`	Pointer to the configuration information.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance has not been assigned.
`TMLIBAPP_ERR_NOT_SETUP`	Instance has not been set up.
`TMLIBAPP_ERR_INVALID_COMMAND`	Unrecognized command.
`VT_ERR_ICP_REQUEST_IN_PROGRESS`	The instance has an ICP request on the queue.

Description

This function will configure specific Video Transformer instance parameters. It uses the standard **tsaControlArgs_t** structure defined in the tsa.h include file to specify the configuration information. The control structures **command** field should be set to one of the values specified by **tmalVtransICPConfigTypes_t** with certain restrictions. The following table specifies the legal **command** field values with the associated **parameter** pointer field:

Command	Parameter
VTRANS_CONFIG_OVERLAY_ENABLE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_BITMASK_ENABLE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_DEINTERLACE_ENABLE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_ANTIFLICKER_ENABLE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_USE_DSPCPU_DEINTERLACE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_NOT_CONSERVATIVE	Pointer to Bool. TRUE = Enabled
VTRANS_CONFIG_OUTPUT_FORMAT	Pointer to tmVideoFormat_t
VTRANS_CONFIG_OVERLAY_POSITION	Pointer to tmalVtransICPOverlayPosition_t
VTRANS_CONFIG_OVERLAY_ALPHA	Pointer to tmalVtransICPAlpha_t
VTRANS_CONFIG_BITMASK	Pointer to tmalVtransICPBitMaskSetup_t

tmalVtransICPProcessFrame

```
tmLibappErr_t tmalVtransICPProcessFrame(
   Int              instance,
   ptmYuvPacket_t   inputPacket,
   ptmAvPacket_t    overlayPacket,
   ptmAvPacket_t    outputPacket,
   Bool             formatChange
);
```

Parameters

`instance`	Instance value.
`inputPacket`	Pointer to the input frame's packet structure.
`overlayPacket`	Pointer to the overlay frame's packet structure. This may be Null if the overlay is disabled.
`outputPacket`	Pointer to the output frame's packet structure. This may be Null, provided that the instance has setup the **outputPCIAddr** field in the image structure.
`formatChange`	Indicates that the input or overlay format has changed.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance has not been assigned.
`TMLIBAPP_ERR_NOT_SETUP`	Instance has not been set up.
`VT_ERR_OVERLAY_SETUP_REQUIRED`	Overlay has not been set up.
`VT_ERR_BITMASK_SETUP_REQUIRED`	Bitmask has not been set up.
`VT_ERR_INVALID_ADDRESS`	The instance **outputPCIAddr** is Null and the outputPacket is Null.
`VT_ERR_NO_INPUT_PACKET`	No input packet.
`VT_ERR_NO_OVERLAY_PACKET`	No overlay packet.
`VT_ERR_NO_OUTPUT_PACKET`	No output packet.
`VT_ERR_IMAGE_FORMAT`	Input format is not YUV.
`VT_ERR_OVERLAY_FORMAT`	Overlay format is not valid.
`VT_ERR_OUTPUT_FORMAT`	Output format is not valid.
`VT_ERR_NO_MORE_NODES`	ICP queue is full.
`VT_ERR_ICP_REQUEST_IN_PROGRESS`	The instance has an ICP request on the queue.

Description

This function is used to place a video transformation request on the ICP queue. If the formatChange flag is set, then the respective fields in the instance input and overlay structures are updated; otherwise the current instance values are used. The required ICP operations to satisfy the input, overlay, and output conditions are then determined. An ICP request is then placed on the queue, and the progress function is called to indicate this.

The ICP processing is asynchronous, and the caller is notified when the request has completed via the completion function. This completion function is specified by the user during the instance setup.

Each instance can only have one request on the ICP request at any instant of time. Therefore, after calling the **tmalVtransICPProcessFrame** function, the user must wait until the completion function is executed before calling the **tmalVtransICPProcessFrame** function again.

tmolVtransICPGetCapabilities

```
tmLibappErr_t tmolVtransICPGetCapabilities(
   ptmolVtransICPCapabilities_t   *pcap
);
```

Parameters

`pcap`	Pointer to a variable in which to return a pointer to capabilities data.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function is provided so that a system resource controller can find out about the Video Transformer library before installing. Fills in the address of the capabilities structure. By default, the OL layer queries the AL layer for its capabilities.

tmolVtransICPOpen

```
tmLibappErr_t tmolVtransICPOpen(
   Int   *instance
);
```

Parameters

`instance`	Pointer to the (returned) instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation failed while creating the instance.
`VT_ERR_NO_FREE_INST`	The maximum number of transformers are already allocated.
`VT_ERR_DEVICE_LIBRARY_ERROR`	Some other process is already using the ICP.

Description

Creates an instance of a Video Transformer. This will automatically call the **tmalVtrans-ICPOpen** function.

tmolVtransICPClose

```
tmLibappErr_t tmolVtransICPClose(
   Int   instance
);
```

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance variable is not valid.
`TMLIBAPP_ERR_NOT_STOPPED`	Instance has not been stopped.
`TMLIBAPP_ERR_DEVICE_LIBRARY_ERROR`	The ICP library returned an error.

Description

This function will shut down an instance of the video transformer. The instance must be stopped (i.e. not streaming data) before this function is called; this is done by calling **tmolVtransICPStop** first.

tmolVtransICPGetInstanceSetup

```
tmLibappErr_t tmolVtransICPGetInstanceSetup(
   Int                              instance,
   ptmolVtransICPInstanceSetup_t   *setup
);
```

Parameters

`instance`	The instance.
`setup`	Pointer to a variable in which to return a pointer to the setup data.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function returns a pointer to the instances setup structure.

tmolVtransICPInstanceSetup

```
tmLibappErr_t tmolVtransICPInstanceSetup(
   Int                              instance,
   tmolVtransICPInstanceSetup_t   *setup
);
```

Parameters

`instance`	The instance.
`setup`	Pointer to the setup structure.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function initializes the Video Transformer instance to the values specified in the setup structure. After this function has been called, any further changes to the instance variables should be made using the **tmolVtransICPInstanceConfig** function.

tmolVtransICPInstanceConfig

```
tmLibappErr_t tmolVtransICPInstanceConfig(
   Int                instance,
   ptsaControlArgs_t  args
);
```

Parameters

`instance`	Instance.
`args`	Pointer to the configuration information.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NOT_STARTED`	The instance has not been started.

Description

This function will configure specific Video Transformer instance parameters. It uses the standard **tsaControlArgs_t** structure defined in the tsa.h include file to specify the configuration information. The control structures **command** field should be set to one of the values specified by **tmalVtransICPConfigTypes_t**. Refer to the **tmalVtransICPInstanceConfig** function for information concerning the arguments which need to be passed.

Note
This function can only be called if the video transformer is currently streaming data.

tmolVtransICPStart

```
tmLibappErr_t tmolVtransICPStart(
   Int   instance
);
```

Parameters

instance	The instance.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_ALREADY_STARTED	The instance is already streaming data.

Description

This function starts the data streaming operation of the Video Transformer. It creates an instance of the Video Transformer task and starts it.

tmolVtransICPStop

```
tmLibappErr_t tmolVtransICPStop(
   Int   instance
);
```

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_ALREADY_STOPPED`	The instance is not streaming.

Description

This function stops the data streaming operation of the Video Transformer.

Chapter 8

TriMedia Motion JPEG Decoder (VdecMjpeg) 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.

Motion JPEG Decoder API Overview

Motion JPEG (MJPEG) is an implementation of JPEG for a sequence of video frames. As of now no known industry-wide standard exists. The MJPEG Decoder Library (VdecMjpeg) available with this release implements standard JFIF (JPEG File Interchange Format File), Motion JPEG format A (MJPEG-A) and Motion JPEG format B (MJPEG-B) decoding for baseline sequentially encoded frames. The current implementation of MJPEG decoder supports bitstreams with:

- LossyJPEG compression (DCT plus Huffman run length encoder).
- Bit stream from input images with 8-bit precision.
- Image formats: Monochrome and YCbCr formats (4:2:2 and 4:1:1).

See Pennebaker & Mitchell, "JPEG Still Image Data Compression Standard", Van Nostrand-Reinhold NY, 1993 for more details on JPEG and "Motion JPEG Format", Draft 2, April 15 1996, courtesy Apple Computer Inc. for details on MJPEG Formats A and B.

MJPEG-A is in full compliance with the ISO JPEG specification. Each frame contains two fields, with the first one being the odd field. Each field is a standard JPEG stream. More than one frame in a file makes a MJPEG Sequence file. In addition to standard JFIF markers (JPEG file interchange format), MJPEG-A adds a new application marker called APP1 (id = "ff e1"). MJPEG-B is nothing but Motion JPEG A stripped of all markers.
The various fields of the APP1 marker are given below:

1. Unused: typically 0000
2. Tag: It should contain "mjpeg"
3. Field size: size of image data
4. Padded field size
5. Offset to next field
6. DCT Quantization table offset
7. Huffman Table Offset
8. Start of Image Offset
9. Start of Scan Offset
10. Start of Data Offset

All fields are 4 bytes and in Big Endian order.

This library provides a standard set of seven APIs, like other TriMedia components that conform to the TriMedia Software Streaming Architecture (TSSA). All interfaces and data structures are fully compliant with this architecture. The component takes a stream of tmAvPackets as input and produces a stream of tmAvPackets (YUV data) at the output.

The Application Library component (tmalVdecMjpeg) provides the basic functionality of OS independent JPEG decoding, while the operating system application library (OL) component (tmolVdecMjpeg) takes care of all inputs and outputs.

Performance

The typical performance obtained is around 8Mbits/sec of encoded bit stream, as measured on a cycle accurate simulator with known software tuning on a 100 MHz tml processor

Demonstration Programs

The VdecMjpeg component is normally used through the OL layer (tmolVdecMjpeg). An example program exolVdecMjpeg.c is provided to illustrate the use of this component. It takes an MJPEG file as an input stream, decodes it and puts it to VO. It uses the File Reader component to open and stream the data as packets to the VdecMjpeg component. After processing, the VdecMjpeg component streams packets of Yuv data to the VrendVO component which would in turn put it onto the VideoOut.

Overview of the tmolVdecMjpeg / tmalVdecMjpeg Component

The tmolVdecMjpeg component layer takes care of properly passing OS dependent parameters like empty and full Queue IDs to the tmalVdecMjpeg component. Default API's are provided and the API implementation is also largely the same as the default implementations provided with tsaDefaults.c.

The tmalVdecMjpeg component library provides an interface consistent with TSSA and provides in all, six C callable functions. A typical Usage Sequence will be:

1. tmalVdecMjpegGetCapabilities to get the decoder capabilities data structure.
2. tmalVdecMjpegOpen. This opens an instance of the decoder. The decoder does not put any restriction on the number of instances.
3. tmalVdecMjpegInstanceSetup. This registers the setup parameters provided by the user into internal instance variables.
4. tmalVdecMjpegStart. This decodes MJPEG frames sequentially until tmalVdecMjpegStop is called.
5. tmalVdecMjpegStop. This changes the **state** variable to STOP.
6. tmalVdecMjpegClose. This invalidates the instance and frees all memory created by the component.

Input Description

The MJPEG Decoder always operates in data streaming mode. It requests packets of data (default size 4K) using the **datain** callback function registered at the time of setup. Input packets are of the type **tmAvPacket_t**. Packet requests are made from within the **tmalVdecMjpegStart**.

The first packet received by the component should be aligned to a MJPEG Chunk. Three types of MJPEG Chunks are recognized by the decoder

- JFIF
- Motion JPEG A
- Motion JPEG B

The first two bytes of JFIF and MJPEG-A formats are “ff” and “d8.”

Output Description

Output packets are of the type **tmAvPacket_t**.

The sizes of the image that is being decoded are embedded within the input stream. In order to create the necessary buffers for the AvPackets and in order to set up the renderer, image sizes and format are to be communicated back to the user. To do this, the user creates a variable of type **ptmalVdecMjpegImageDescription_t** and registers it through the setup variable. The VdecMjpeg component will update this variable immediately after decoding the image description. The first **datain** call for an empty output packet will occur after this. It is the responsibility of the user to have created the buffers before passing the empty packets to the component. Typically the user will poll the **Initialized** field of **ImageDescription** to find out whether the MJPEG Decoder has decoded the image sizes. The user then creates the buffers and puts them into the empty queue. The VdecMjpeg uses these buffers to fill decoded data and puts them out through the **dataout** function.

Stopping the VdecMjpeg Component

The **VdecMjpeg** component may be stopped by either changing the **MjpegStates** variable to **MJPEG_STOP** from the AL layer or by calling tmolVdecMjpegStop. The former will stop the component after the current frame is processed. The latter is implemented by a call to the tsaDefaultStop. The example exolVdecMjpeg.c illustrates one way of stopping the processing chain. When the VdecMjpeg stops, it calls its completion function which may be used to synchronize with the other components.

Motion JPEG Decoder API Data Structures

This section describes all the data structures concerned with the VdecMjpeg component

tmalVdecMjpegStates_t

```
typedef enum {
   MJPEG_RUN,
   MJPEG_STOP,
   MJPEG_PAUSE,
   MJPEG_SKIP
} tmalVdecMjpegStates_t,*ptmalVdecMjpegStates_t;
```

Fields

`MJPEG_RUN`	Value of the instance variable's **state** field when the decoder is decoding the stream. **tmalVdecMjpegStart** puts the component into this state.
`MJPEG_STOP`	Value of the **state** field when the decoder is stopped or is required to be stopped at the end of the current frame.
`MJPEG_PAUSE`	Reserved for future use.
`MJPEG_SKIP`	Value of the **state** field when the user wants to skip the current frame. It is the user's responsibility to release SKIP and put back RUN or STOP. This can be done by using the **progress** function.

tmalVdecMjpegStream_t

```
typedef enum {
   MJPEG_A,
   MJPEG_B,
   MJPEG_JFIF,
   MJPEG_UNSUPPORTED_STREAM_TYPE
} tmalVdecMjpegStream_t;
```

Fields

`MJPEG_A`	Encoded streasm type is Motion JPEG-A.
`MJPEG_B`	Encoded stream type is Motion JPEG-B.
`MJPEG_JFIF`	Encoded stream type is JFIF.
`MJPEG_UNSUPPORTED_STREAM_TYPE`	Unknown input stream.

Description

These are the stream types used by the decoder internally. This is passed to the user through the (**ptmalVdecMjpegImageDescription_t**) ImageDescription field, of the setup variable. Necessary control action can be initiated by the user.

tmalVdecMjpegCapabilities_t, tmolVdecMjpegCapabilities_t

```
typedef struct{
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmalVdecMjpegCapabilities_t, *ptmalVdecMjpegCapabilities_t;
```

Fields

`defaultCapabilities`	Pointer to **tsaDefaultCapabilities_t**.

Description

See tsa.h for details. Replace “al” by “ol” in the structure above to get **tmolVdecMjpeg-Capabilities_t**.

tmalVdecMjpegImageDescription_t

```
typedef struct{
   Int32                   ImageHeight;
   Int32                   ImageWidth;
   Int32                   ImageStride;
   Int32                   PaddedImageHeight;
   tmVideoRGBYUVFormat_t   ImageFormat;
   Bool                    Initialized;
} tmalVdecMjpegImageDescription_t, *ptmalVdecMjpegImageDescription_t;
```

Fields

`ImageHeight`	Actual image height.
`ImageWidth`	Actual image width.
`ImageStride`	Calculated width of the Image Buffer based on the required granularity of Output Stride. See **tmalVdecMjpegInstanceSetup_t**.
`PaddedImageHeight`	Image height for which the bit stream is encoded. This can be larger than **ImageHeight** to take care of image heights which are not a multiple of eight.
`ImageFormat`	Decoded image format, one of vdfMono, vdfYUV420Planar, or vdfYUV422Planar.
`Initialized`	Set to True after the decoder fills in the other fields.

Description

This is the image description extracted from the encoded stream.

Note
The user is expected to create the image buffers and pass it to the decoder. The expected size of the buffer is the product of **PaddedImageHeight** times **ImageStride**. The user can poll the Initialized field to know when to create these buffers.

tmalVdecMjpegInstanceSetup_t, tmolVdecMjpegInstanceSetup_t

```
typedef struct{
   ptsaDefaultInstanceSetup_t         default_setup;
   Bool                               littleEndian;
   Int32                              granularityOfOutputAddress;
   Int32                              granularityOfOutputStride;
   ptmalVdecMjpegStates_t             state;
   ptmalVdecMjpegImageDescription_t   ImageDescription;
} tmalVdecMjpegInstanceSetup_t, *ptmalVdecMjpegInstanceSetup_t;
```

Fields

`default_setup`	Pointer to a **tsaDefaultInstanceSetup_t** variable. See tsa.h.
`littleEndian`	True if the component is required to work in little endian mode, otherwise it is False. Currently only the compile time selection is employed.
`granularityOfOutputAddress`	The address alignment required for the output decoded image buffers (Y). Typically 64 for **Vrend** and 128 for **Vtrans**.
`granularityOfOutputStride`	Image width alignment due to output hardware constraints. Typically Nil for **Vrend** and 64 for **Vtrans**.

Description

This is the **InstanceSetup** struct for the MJPEG decoder. Replace “al” by “o” in information above to get **tmolVdecMjpegInstanceSetup_t**.

tmalVdecMjpegProgressFlags_t

```
typedef enum{
   MJPEG_REPORT_FORMAT,
   MJPEG_REPORT_FIELD,
   MJPEG_REPORT_FRAME
   MJPEG_REPORT_STOP,
   MJPEG_REPORT_EOF
} tmalVdecMjpegProgressFlags_t, *ptmalVdecMjpegProgressFlags_t;
```

Fields

`MJPEG_REPORT_FORMAT`	Causes the **progress** function to be called immediately after the decoder extracts **ImageDescription** from the stream.
`MJPEG_REPORT_FIELD`	Causes the **progress** function to be called after each field has been decoded.
`MJPEG_REPORT_FRAME`	Causes the **progress** function to be called after each frame has been decoded.
`MJPEG_REPORT_STOP`	Causes the **progress** function to be called while stopping.
`MJPEG_REPORT_EOF`	Causes the **progress** function to be called when dataSize of the input packrt is lesser than the bufSize. This can be treated as a warning for an end of input stream.

Motion JPEG Decoder API Functions

This section describes the various API functions for the VdecMjpeg component.

tmalVdecMjpegOpen, tmolVdecMjpegOpen

```
tmLibappErr_t tmalVdecMjpegOpen(
   Int   *instance
);
```

Parameters

`instance`	Pointer to the instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Unable to allocate memory for decompressor.

Description

This function will create an instance of the VdecMjpeg component.

tmalVdecMjpegStart, tmolVdecMjpegStart

```
tmLibappErr_t tmalVdecMjpegStart(
   Int   instance
);
```

Parameters

`instance`	Instance value assigned at the call of **the MJPEG-Open** function.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_DATAIN_FAILED`	Returned if **datain** callback function fails.
`TMLIBAPP_ERR_DATAOUT_FAILED`	Returned if **dataout** callback function fails.
`MJ_ERR_INVALID_PACKET`	Returned if the input or output packets received through the empty/full queues are of improper format or size.
`MJ_ERR_CORRUPT_STREAM`	Returned if the Huffman encoded bitstream yields invalid states or symbols.

Description

This function will sequentially decode all frames from a MJPEG file.

tmalVdecMjpegStop, tmolVdecMjpegStop

```
tmLibappErr_t tmalVdecMjpegStop(
   Int   instance
);
```

Parameters

`instance`	Instance value assigned at the call of **the VdecMjpeg Open** function.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Returned if the instance has not been opened.
`TMLIBAPP_ERR_NOT_SETUP`	Returned if the instance has not been set up.

Description

tmalVdecMjpegStop merely changes the components **state** variable to STOP.

tmolVdecMjpegStop calls tsaDefaultStop.

tmalVdecMjpegClose, tmolVdecMjpegClose

```
tmLibappErr_t tmalVdecMjpegClose(
   Int   instance
);
```

Parameters

`instance`	Instance value assigned at the call of **the MJPEG-Open** function.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The passed parameter is not a valid instance.

Description

This function invalidates the instance and frees all memory allocated for instance variables and Decompression Instance.

tmalVdecMjpegGetCapabilities, tmolVdecMjpegGetCapabilities

```
tmLibappErr_t tmalVdecMjpegGetCapabilities(
   ptmalVdecMjpegCapabilities_t   *capabilities
);
```

Parameters

`capabilities`	Pointer to variable in which to return a pointer to capabilities data.

Return Codes

`TMLIBAPP_OK`	Success.
`MJ_ERR_NULL_POINTER`	The capbilities pointer is Null.

Description

This function initializes the **capabilities** struct with the MJPEG Decoder component's values.

tmalVdecMjpegInstanceSetup, tmolVdecMjpegInstanceSetup

```
tmLibappErr_t tmalVdecMjpegInstanceSetup(
   Int                              instance,
   ptmalVdecMjpegInstanceSetup_t    setup
);
```

Parameters

`instance`	Instance value assigned at the call of **the MJPEG-Open** function.
`setup`	Pointer to the **setup** data struct.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance value is not valid.
`TMLIBAPP_ERR_INVALID_SETUP`	A field in the **setup** data struct is invalid.

Description

This function registers the setup parameters provided by the user into the internal instance variables.

tmolVdecMjpegGetInstanceSetup

```
tmLibappErr_t tmolVdecMjpegGetInstanceSetup(
   Int                              instance,
   ptmolVdecMjpegInstanceSetup_t    setup
);
```

Parameters

`instance`	The instance.
`setup`	Pointer to the Instance setup structure of **VdecMjpeg**.

Return Codes

`TMLIBAPP_OK`	Success

Description

This function assigns the pointer to **tmolVdecMjpegInstanceSetup_t** structure allocated by the MJPEG Open function to setup.

Chapter 9

TriMedia Motion JPEG Encoder (VencMjpeg) 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.

Motion JPEG Encoder API Overview

Motion JPEG (MJPEG) is an implementation of JPEG for a sequence of video frames. As of now no known official standard exists. The MJPEG Encoder Library provided with this release implements standard JFIF (JPEG File Interchange Format File), motion JPEG format A (MJPEG-A) and motion JPEG format B (MJPEG-B) baseline sequential encoding for frames. The encoder encodes bitstreams using

- Lossy, DCT based transform, followed by Huffman run length encoding.
- Input precision of 8 bits
- YCbCr 4:2:2 or YUV 4:1:1 image format.

See Pennebaker & Mitchell, *JPEG Still Image Data Compression Standard,* Van Nostrand-Reinhold NY, 1993 for more details on JPEG, and *Motion JPEG Format,* Draft 2, April 15, 1996, courtesy Apple Computer Inc., for details on MJPEG Formats A and B (See http://www.QuickTimeFAQ/developer/).

MJPEG-A is in full compliance with the ISO JPEG specification. Each frame contains two fields, with the first one being the top field. Each field is a standard JPEG stream. More than one frame in a file makes an MJPEG Sequence file. In addition to standard JFIF markers (JPEG file interchange format), MJPEG-A adds a new application marker called APP1 (id = ”ffe1”). MJPEG-B is nothing but motion JPEG A stripped of all markers. The various fields of the APP1 marker are given below (All fields are 4 bytes long and are in Big Endian order):

1. Unused: typically 0000
2. Tag: It should contain “mjpeg”
3. Field size: size of encoded image data for each field
4. Padded field size
5. Offset to next field
6. DCT Quantization table offset
7. Huffman Table Offset
8. Start of Image Offset
9. Start of Scan Offset
10. Start of Data Offset

All fields are 4 bytes and in Big Endian order.

This library provides a standard set of seven APIs, like other tm components, which conform to the TriMedia Software Streaming Architecture (TSSA). All interfaces and data structures are fully compliant with this architecture. The component takes a stream of tmAvPackets (YUV data) as input and produces a stream of tmAvPackets at the output.

The Application Library component (tmalVencMjpeg) provides the basic functionality of OS independent JPEG encoding, while the operating system application library (OL) component (tmolVencMjpeg) takes care of all inputs and outputs.

Performance

The typical performance obtained is around 4Mbits/sec of encoded bitstream, as measured on a cycle-accurate simulator with known software tuning on a 100 MHz TM-1 processor.

Demonstration Programs

The VencMjpeg component is normally used through the OL layer (tmolVencMjpeg). An example program exolVencMjpeg.c is provided to illustrate the use of this component. It takes a tmAvPacket_t (YUV data) as an input stream, encodes it and puts it into a file. It uses the VdigVI component to capture the images and stream the data as packets to the VencMjpeg component. After processing, the VencMjpeg component streams packets to the Fwrite component which would in turn put it into the file.

Overview of the tmolVencMjpeg / tmalVencMjpeg Component

The tmolVencMjpeg component layer takes care of properly passing OS dependent parameters like empty and full Queue IDs to the tmalVencMjpeg component. Default API's are provided and the API implementation is also largely the same as the default implementations provided with tsaDefaults.c.

The tmalVencMjpeg component library provides an interface consistent with TSSA and provides in all, six C callable functions. A typical Usage Sequence will be:

1. tmalVencMjpegGetCapabilities to get the encoder capabilities data structure.
2. tmalVencMjpegOpen. This opens an instance of the encoder. The encoder does not put any restriction on the number of instances.
3. tmalVencMjpegInstanceSetup. This registers the setup parameters provided by the user into internal instance variables.
4. tmalVencMjpegStart. This encodes YUV data frames sequentially until tmalVencMjpegStop is called.
5. tmalVencMjpegStop. This changes the **state** variable to STOP.
6. tmalVencMjpegClose. This invalidates the instance and frees all memory created by the component.

Input Description

The MJPEG Encoder requests data using the datain callback function registered at the time of setup. Input packets are of type tmAvPacket_t (YUV data). Packet requests are made from within the tmalVencMjpegStart.

The sizes of the image that has to be encoded are present in the **tmalVencMjpegImage-Description_t** structure registered during setup of the instance.

Output Description

Output packets are of type **tmAvPacket_t**. The user creates the buffers and puts them into the empty queue. The VencMjpeg uses these buffers to fill encoded data and puts them out through the dataout function.

The output stream will be in any of the following formats:

- JFIF
- Motion JPEG A
- Motion JPEG B

Stopping the VencMjpeg Component

The **VencMjpeg** component may be stopped by calling tmalVencMjpegStop or tmolVenc-MjpegStop. The former will stop the component after the current frame is processed. The latter is implemented by a call to the tsaDefaultStop. The example exolVencMjpeg.c illustrates one way of stopping the processing chain. When the VencMjpeg stops, it calls its completion function which may be used to synchronize with the other components.

Motion JPEG Encoder API Data Structures

This section presents all the data structures concerned with the VencMjpeg component

tmalVencMjpegStates_t

```
typedef enum {
   MJPGEn_RUN   = 0x1,
   MJPGEn_STOP  = 0x2,
   MJPGEn_PAUSE = 0x4,
   MJPGEn_SKIP  = 0x8,
}tmalVencMjpegStates_t,*ptmalVencMjpegStates_t;
```

Fields

`MJPGEn_RUN`	Value of the instance variable's **state** field when the encoder is encoding the stream. **tmalVencMjpegStart** puts the component into this state.
`MJPGEn_STOP`	Value of the **state** field when the encoder is stopped or is required to be stopped at the end of the current frame.
`MJPGEn_PAUSE`	Reserved for future use.
`MJPGEn_SKIP`	Value of the **state** field when the user wants to skip the current frame. It is the user's responsibility to release SKIP and put back RUN or STOP. This can be done by using the **progress** function.

tmalVencMjpegStream_t

```
typedef enum {
   MJPGEn_A                        = 0x1,
   MJPGEn_B                        = 0x2,
   MJPGEn_JFIF                     = 0x4,
   MJPGEn_UNSUPPORTED_STREAM_TYPE  = 0x8,
}tmalVencMjpegStream_t,*ptmalVencMjpegStream_t;
```

Fields

`MJPGEn_A`	Encoded stream type is Motion JPEG-A.
`MJPGEn_B`	Encoded stream type is Motion JPEG-B.
`MJPGEn_JFIF`	Encoded stream type is JFIF.
`MJPGEn_UNSUPPORTED_STREAM_TYPE`	Unknown input stream.

Description

These are the stream types generated by the encoder. This is passed by the user through the image description field of the setup variable.

tmalVencMjpegProgressFlags_t

```
typedef enum {
   MJPGEn_REPORT_START_ENCODING    = 0x1,
   MJPGEn_REPORT_STOP_ENCODING     = 0x2,
}tmalVencMjpegProgressFlags_t,*ptmalVencMjpegProgressFlags_t;
```

Fields

`MJPGEn_REPORT_START_ENCODING`	Report before starting the encoding.
`MJPGEn_REPORT_STOP_ENCODING`	Report after encoding has stopped.

tmalVencMjpegBufferType_t

```
typedef enum {
   FULL_BUFFER   = 0x1,
   PADDED_FF     = 0x2,
} tmalVencMjpegBufferType_t,*ptmalVencMjpegBufferType_t;
```

Fields

FULL_BUFFER	Output buffer allocated equals the maximum image size.
PADDED_FF	Output buffer allocated is of fixed size.

tmalVencMjpegCapabilities_t, tmolVencMjpegCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmalVencMjpegCapabilities_t,*ptmalVencMjpegCapabilities_t;
```

Fields

defaultCapabilities	Pointer to **tsaDefaultCapabilities_t**

Description

See tsa.h for details. Replace “al” by “ol” in information above to get description of **tmolVencMjpegCapabilities_t**.

tmalVencMjpegImageDescription_t, tmolVencMjpegImageDescription_t

```
typedef struct {
   UInt32                  height;
   UInt32                  width;
   UInt32                  imageStride;
   UInt32                  is_field;
   tmVideoRGBYUVFormat_t   format;
   tmalVencMjpegStream_t   stream;
}tmalVencMjpegImageDescription_t,*ptmalVencMjpegImageDescription_t;
```

Fields

`height`	This must contain the height of the input image.
`width`	This must contain the width of the input image.
`imageStride`	This must contain the stride of the input image.
`is_field`	This must be zero for frame based encoding and 1 for field based encoding.
`format`	This must contain the input image format which is YUV 4:2:2 or YUV 4:1:1.
`stream`	This must contain the output stream type which can be JFIF, MJPEG A or MJPEG B.

Description

This contains parameters that describe the image. It also contains a field to indicate the type of the output stream to be generated.

Note
The user is expected to create the image buffers and pass them to the encoder. The expected size of the buffer is the product of the height and imageStride.

tmalVencMjpegInstanceSetup_t/ tmalVencMjpegInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t        defaultSetup;
   ptmalVencMjpegImageDescription_t  ImageDescription;
   tmalVencMjpegBufferType_t         BufferType;
}tmalVencMjpegInstanceSetup_t,*ptmalVencMjpegInstanceSetup_t;
```

Fields

`defaultSetup`	Stores the default values of this application library.
`ImageDescription`	This describes the image parameters.
`BufferType`	This indicates whether the output data will have padded data or not. This depends on the memory available with the application.

Description

This structure contains all the required parameters to initialize the MJPEG video encoder. Replace “al” by “ol” in information above to get description of **tmolVencMjpegInstanceSetup_t**.

Motion JPEG Encoder API Functions

This section presents the various API functions for the MJPEG Encoder component.

tmalVencMjpegGetCapabilities / tmolVencMjpegGetCapabilities

```
tmLibappErr_t tmalVencMjpegGetCapabilities (
   ptmalVencMjpegCapabilities_t   *cap
)
tmLibappErr_t tmolVencMjpegGetCapabilities (
   ptmalVencMjpegCapabilities_t   *cap
)
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`MJ_ERR_NULL_POINTER`	The capabilities pointer is Null.

Description

This function initializes the **cap** struct with the MJPEG Encoder component's values.

tmalVencMjpegOpen / tmolVencMjpegOpen

```
tmLibappErr_t tmalVencMjpegOpen(
   Int   *instance
);
tmLibappErr_t tmolVencMjpegOpen(
   Int   *instance
);
```

Parameters

Pointer to the instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Unable to allocate memory for compressor.

Description

This function will create an instance of the **VencMjpeg** component.

tmalVencMjpegClose / tmolVencMjpegClose

```
tmLibappErr_t tmalVencMjpegClose(
   Int   instance
);
tmLibappErr_t tmolVencMjpegClose(
   Int   instance
);
```

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The passed parameter is not a valid instance.

Description

This function invalidates the instance and frees all memory allocated for instance variables and Decompression Instance.

tmolVencMjpegGetInstanceSetup

```
tmLibappErr_t tmolVencMjpegGetInstanceSetup(
   Int                              instance,
   ptmolVencMjpegInstanceSetup_t    setup
);
```

Parameters

`instance`	Instance value assigned at the call of the MJPEG Open function.
`setup`	Pointer to the Instance setup structure of **VencMjpeg**.

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function assigns the pointer to **tmolVencMjpegInstanceSetup_t** structure allocated by the MJPEG Open function to setup.

tmalVencMjpegInstanceSetup / tmolVencMjpegInstanceSetup

```
tmLibappErr_t tmalVencMjpegInstanceSetup (
   Int                              instance,
   ptmalVencMjpegInstanceSetup_t    instanceSetup
);
tmLibappErr_t tmolVencMjpegInstanceSetup(
   Int                              instance,
   ptmalVencMjpegInstanceSetup_t    instanceSetup
);
```

Parameters

`instance`	Instance value assigned at the call of the MJPEG Open function.
`instanceSetup`	Pointer to the instanceSetup data structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance value is not valid.
`TMLIBAPP_ERR_INVALID_SETUP`	A field in the setup data struct is invalid.

Description

This function registers the setup parameters provided by the user into the internal instance variables.

tmalVencMjpegStart / tmolVencMjpegStart

```
tmLibappErr_t tmalVencMjpegStart(
   Int   instance
)
tmLibappErr_t tmolVencMjpegStart(
   Int   instance
)
```

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_DATAIN_FAILED`	The **datain** callback function failed.
`TMLIBAPP_ERR_DATAOUT_FAILED`	The **dataout** callback function failed.

Description

This function does the encoding of the data in a streaming mode.

tmalVencMjpegEncodeFrame

```
tmLibappErr_t tmalVencMjpegEncodeFrame(
   Int   instance
)
```

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_DATAIN_FAILED`	The **datain** callback function failed.
`TMLIBAPP_ERR_DATAOUT_FAILED`	The **dataout** callback function failed.

Description

This function does the encoding of the data in a push mode, one packet at a time.

tmalVencMjpegStop, tmolVencMjpegStop

```
tmLibappErr_t tmalVencMjpegStop(
   Int   instance
)
tmLibappErr_t tmolVencMjpegStop(
   Int   instance
)
```

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance has not been opened.
`TMLIBAPP_ERR_NOT_SETUP`	The instance has not been set up.

Description

The function merely changes the component's **state** variable to STOP. The function calls **tsaDefaultStop**.

Chapter 10

Natural Motion Video Transformer (VtransNM) 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.

VtransNM API Overview

The natural motion video transformer is a high-quality video transformer that transforms standard interlaced signals, such as PAL and NTSC, to a progressive signal. It does this using either smart deinterlacing techniques or heuristic motion estimation and compensation techniques tailored to execution in software.

Before starting, VtransNM optionally does a film mode detection algorithm, which detects 3:2 and 2:2 film modes, or regular interlaced broadcasts. Then it does a motion estimation processing step and then a deinterlacing step. The details of the algorithm are proprietary and not disclosed.

There are two distinct processing techniques used in the deinterlacer. The first category has three options of deinterlacing, in which there is no motion compensation performed. The most sophisticated of these three techniques is the default, known as *film detection*. This mode detects film material (originally 24 frames per second) and has a state-of-the-art deinterlacer using previous and next fields. The result is a very sharp progressive image. The cheaper modes are *field insertion* and *field insertion with median filtering*. The second category is a heuristic in which motion vectors are determined and the deinterlacer generates the missing fields by interpolation taking the motion vectors into account. This mode is bound by processing and memory bandwidth requirements. The heuristic is very good, but on some images where small objects move around the motion estimator is too coarse, which can lead to artifacts. The positive effects of this mode are best noticeable on scenes where a camera pans. The motion judder introduced by the 3:2 pull down is completely compensated and the result is very smooth. This mode is not the default since there are artifacts introduced by the heuristic.

The CPU requirements are input-dependent, but average around 100 MHz for full screen NTSC, but the maximum computing requirements can exceed the processing power of a 125 MHz TM-1100. In this rare case, a field skip may occur. There are two computationally inexpensive processing modes, in which the input/output behavior of the program is the same but the amount of processing cycles needed is greatly reduced and fixed. These modes are the field insertion and median filtering modes. They can be used when an application needs more CPU power at the cost of reduced image quality. The field insertion mode needs approximately 31 MHz of a TM-1100 and the median filter approximately 38 MHz. Switching between processing modes does not introduce any artifacts in the video processing chain.

The delay within the Natural Motion component is 4 fields. That is, after the end of the captured field, there are four field captures before the 'deinterlaced frame' is presented.

The VtransNM module is to be used with a PLL that sets the video-out clock such that the video buffers, i.e., the buffers that contain the actual video frames, are released just before the video-in component needs them again. Since memory is a scarce resource, this PLL is required. The PLL is not part of the VtransNM component and should be supplied by the application. As long as the buffer returned by the output chain is in time to

be passed to the input chain VtransNM has enough time and buffers to do its work. When the PLL is not supplied unexpected results may occur.

Note
The name Natural Motion suggests the deinterlacer does motion compensation, which is not true for all modes. VtransNM has as default mode a mode that does not do motion compensated deinterlacing. The default mode is suggested for products using the current VtransNM library.

Note
VtransNM needs a software PLL that locks the output clock to the input clock. The PLL should be such that buffers released by the output are just in time to be passed to the input digitizer. This PLL is application dependent and therefore not integrated in any TSSA component.

Limitations

Because Natural Motion requires a lot of memory, special care is taken for the memory allocation. The VtransNM component implements it's own buffer management for the video buffers. The regular TSSA interface would lead to too much copying of data and would strain the algorithm's computational and memory requirements. These requirements lead to several unexpected side effects. For instance, TSSA packets do not have the same buffer pointers all the time, because VtransNM reassigns buffers to packets. VtransNM also keeps inspecting the data in the packets that are already sent out to the output queue, so it expects that these packets are read only by the components in the output chain. On closing the component VtransNM does not free the memory that was allocated for the buffers. All internal memory is released, but buffers in the empty input queue need to be released by the application because VtransNM does not know if these buffers are in use.

It is possible to let the application allocate the buffers. In this case a buffer pointer can be passed to the component that points to a preallocated buffer big enough to create all the smaller video buffers from it. In this situation, VtransNM does not allocate new memory but simply slices the bigger buffer into cache-aligned video buffers. The **tmolVtransNMGetInstanceSetup** function will return the total number of video buffers VtransNM needs. On **tmolVtransNMStop**, the memory will not be used until **tmolVtransNMStart** is called. The memory can be freed by the application when **tmolVtransNMClose** is called. This allows the application to reuse the memory when Natural Motion is stopped.

The buffer reuse scheme requires special care to be taken at **tmolVtransNMStop**. The buffers are reference counted, and buffers sent to the output chain have a non-zero reference count (VtransNM is still inspecting them). That means that buffers returned from the output component need to go through the VtransNM component. Normally on **tmolVtransNMStop** there is no problem, because the buffer will come back in the right queue and will be picked up when the component is restarted. But when VtransNM is stopped and the queue to the output chain is reconnected to another component, the administration will be incomplete if VtransNM is ever started again. The correct way to

Stop VtransNM and then reconnect the output queue is to stop the output chain first such that all packets are released and sent back to the TSSA empty queue. Then Stop VtransNM and all packets are guaranteed to have reference count 0 and reside in the TSSA empty input queue of the VtransNM component.

Note 1
VtransNM still reads out of the buffers sent to its output, and expects them to be unmodified.

Note 2
VtransNM wants to allocate it's own buffers. Deallocation is NOT done by VtransNM, not even on a call to **tmolVtransNMClose**. All buffers are cache aligned. It is possible to pass a pointer to VtransNM from which buffers are created. The buffer passed to VtransNM *cannot* be deallocated until the VtransNM instance is closed. **tmolVtransNMStop** is not enough because internal buffers are still known internally at **tmolVtransNMStop**.

Note 3
Stopping VtransNM does not mean that output queue can be redirected to another component. If you want to do that, then first the output chain of components needs to be stopped such that all outstanding buffers on the output are seen by the VtransNM component. Then VtransNM can be stopped and the queue can be redirected.

VtransNM Inputs and Outputs

Overview

An overview of the inputs and output of the natural motion video transformer is depicted in Figure 12. There is one input, which is a stream of digitized video fields. There is one output, which is the progressive video stream. Via the control input, the component can be switched into certain processing modes, see **tmolVtransNMConfig_t**.

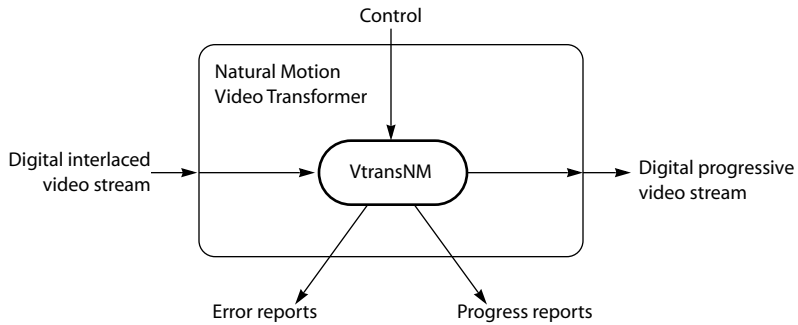


Figure 12 Overview of the VtransNM transformer

Inputs

The capability format for the input descriptor is set to

```
tmAvFormat_t inFormat = {
   sizeof(tmAvFormat_t),              /* size        */
   0,                                 /* hash        */
   0,                                 /* referenceCount */
   avdcVideo,                         /* dataClass   */
   vtfYUV,                            /* dataType    */
   vdfYUV422Planar,                   /* dataSubtype */
   vdfFieldInFrame                    /* description */
};
```

The incoming packets are tmAvPackets, which have the format set to type tmVideoFormat_t. From the packets, the stride is checked against the bufferStride given at **tmolVtransNMGetInstanceSetup** of the VtransNM instance. The activeVideoStartY and activeVideoStartX are taken as starting points in the buffer to start processing. The deinterlacing only applies to the region from activeVideoStart up to the imageHeight field from the instance setup.

The video in, input descriptor has number 0, see **inputDescriptors** in the **tmalDefaultCapabilities_t**.

```
#define VTRANSNM_MAIN_INPUT  0
```

Outputs

The is only one output of the VtransNM component and that is the upconverted video. The output capability format is set to:

```
tmAvFormat_t videoFormat = {
   sizeof(tmAvFormat_t),   /* size           */
   0,                      /* hash           */
   0,                      /* referenceCount */
   avdcVideo,              /* dataClass      */
   vtfYUV,                 /* dataType       */
   vdfYUV422Planar,        /* dataSubtype    */
   vdfProgressive          /* description    */
};
```

The output descriptor assignment is:

```
#define VTRANSNM_MAIN_OUTPUT  0
```

VtransNM Errors

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

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

VtransNM Progress

There are no progress reports produced by VtransNM that calls back into the application at this point. The only progress report used is the **tsaProgressFlagChangeFormat** and this one is handled internally by TSSA internally.

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

VtransNM Configuration

The following control modes can be set via calls to **tmolVtransNMInstanceConfig**:

1. VTRANSNM_CMD_FIELD_INSERTION, switch mode to the computationally least expensive field insertion processing mode.
2. VTRANSNM_CMD_MEDIAN_FILTER, switch mode to field insertion with median filtering.
3. VTRANSNM_CMD_FILM_DETCTION, switch mode to deinterlacing with film detection and heuristic deinterlace filtering. This is the default mode resulting in very sharp progressive output.
4. VTRANSNM_CMD_MOTION_ESTIMATION, switch mode to film detection, motion estimation and deinterlacing with motion compensation.

These mode changes are asynchronous calls, and apply to the next incoming packet. Other configuration modes are triggered by format changes on the input packets.

VtransNM API Data Structures

This section describes the VtransNM component data structures.

tmolVtransNMInstanceSetup_t

```
typedef struct tmolVtransNMInstance {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   ptmolVtransNMConfig_t       vtransNMConfig;
} tmolVtransNMInstanceSetup_t, *ptmolVtransNMInstanceSetup_t;
```

Fields

`defaultSetup`	See TSSA documentation.
`vtransNMConfig`	See **tmolVtransNMConfig_t**.

Description

Data structure passed to **tmolVtransNMInstanceSetup** to describe the input and output connections and other initial values, see **tmolVtransNMConfig_t**.

tmolVtransNMCapabilities_t

```
typedef struct tmalVtransNMCapabilities{
   ptsaDefaultCapabilities_t   defaultCaps;
} tmolVtransNMCapabilities_t, *ptmolVtransNMCapabilities_t;
```

Fields

`defaultCaps`	See TSSA documentation.

Description

For input and output descriptors, see *VtransNM Inputs and Outputs* on page 158. The text section of transformer is about 140 kb, the initialized data section is about 6 kb, there is no bss requirement.

tmolVtransNMConfig_t

```
typedef struct tmolVtransNMConfig {
   UInt32    packetBase
   UInt32    bufferStride;
   UInt32    bufferHeight;
   UInt32    nrofBuffers;
   UInt8    *buffer;
   UInt32    bufferSize;
} tmolVtransNMConfig_t, *ptmolVtransNMConfig_t;
```

Fields

`packetBase` — VtransNM should install on its TSSA created packets (value of packet->header->id) for debugging and packet tracing. Packets are numbered from packetBase and up. This field is similar to the packetBase field of **tsaInOutDescriptorSetup_t**.

`bufferStride` — Fixed stride for the buffers. During the lifetime of this instance the stride cannot be changed. When the application needs to switch between NTSC and PAL the maximum of the two needs to be taken. The UV buffer strides are taken as half the bufferStride rounded up to the next multiple of 64 bytes. These strides need to be taken into account when the buffer is created by the application and passed to VtransNM.

`bufferHeight` — Height of the buffers that need to be allocated. When PAL and NTSC are used in the application the maximum buffer height (576) needs to be taken. When the VBI data needs to be captured in the same buffer those lines need to be added to.

`nrofBuffers` — Returned by **tmolVtransNMGetCapabilities** as info when the application wants to create the memory from which the buffers are created.

`buffer` — Passed to **tmolVtransNMInstanceSetup** when the application wants to do the memory allocation.

`bufferSize` — The size of the buffer allocated by the application (only used when **buffer** is non-Null). The bufferSize is the total size of the buffer, so it should be nrofBuffers times the size of one buffer. One buffer for NTSC processing should have cache aligned Y and UV buffers and the Y buffer should have size 480 (height) × 768 (stride). There is no room for VBI data, and PAL cannot be processed for these buffers. When the stride is made smaller (for instance 704) the application should allocate

enough buffer space to have the UV buffers cache aligned also (half the Y-stride is not a multiple of 64).

Description

Controls the instance setup and is used by the **tmolVtransNMInstanceSetup** function. Applications need to be aware of the different execution modes of the VtransNM component. For instance, when PAL and NTSC need to be processed the buffers need to be big enough to capture the digitized input. When a data slicer needs to process VBI information, this needs to be taken into account also, that is, the lines for the VBI data need to be allocated also. The example program shows some of these issues.

tmolVtransNMErrorFlags_t

n the enum below, **base** is equal to **Err_base_VTRANSNM** (0x130E0000).

```
typedef enum {
/* Fatal errors */
   VTRANSNM_ERR_VIDEO_FORMAT_EXPECTED  base + 0x0001,
   VTRANSNM_ERR_STRIDE_MODIFIED        base + 0x0002,
   VTRANSNM_ERR_PACKETS_ALLOCATED      base + 0x0003,
   VTRANSNM_ERR_UNKNOWN_COMMAND        base + 0x0004,
   VTRANSNM_ERR_BUFFER                 base + 0x0005,
   VTRANSNM_ERR_FORMAT_INCORRECT       base + 0x0006,
   VTRANSNM_ERR_INVALID_PROCESSOR      base + 0x0007,
   VTRANSNM_ERR_ACT_VIDEO_HEIGHT       base + 0x0008,
   VTRANSNM_ERR_ACT_VIDEO_WIDTH        base + 0x0009,
   VTRANSNM_ERR_INVALID_WIDTH          base + 0x000c,
   VTRANSNM_ERR_INVALID_HEIGHT         base + 0x000d,
   VTRANSNM_ERR_INTERNAL_ERROR         base + 0x01ff
} tmalVtransNMErrorFlags_t;
```

Fields

The fields here describe fatal errors.

`VTRANSNM_ERR_VIDEO_FORMAT_EXPECTED`
: The packet format is not of type **tmVideoFormat_t**.

`VTRANSNM_ERR_STRIDE_MODIFIED`
: The stride of the buffers cannot change during the lifetime of the VtransNM instance. At **InstanceSetup** this is passed to the component, after that it cannot change via the packet formats.

`VTRANSNM_ERR_PACKETS_ALLOCATED`
: The IODescriptor passed as input IODescriptor has already allocated packets.

`VTRANSNM_ERR_UNKNOWN_COMMAND`
: The command passed to **tmolVtransNMInstanceConfig** is unknown.

`VTRANSNM_ERR_BUFFER`
: The passed-in buffer was too small. Alignment needs to be taken into account when creating the buffer.

`VTRANSNM_ERR_FORMAT_INCORRECT`
: The format is incorrect; either the stride was changed or there was an incorrect videoStandard. The errorFlags.description field is set to the format.

`VTRANSNM_ERR_INVALID_PROCESSOR`
: This program cannot run on a TM-1000, due to the new instructions of the TM-1100 that it uses.

`VTRANSNM_ERR_ACT_VIDEO_HEIGHT`
: Active video endY – startY exceeds the NTSC or PAL values.

`VTRANSNM_ERR_ACT_VIDEO_WIDTH`
: Active video endX – startX exceeds the NTSC or PAL values.

VTRANSNM_ERR_INVALID_WIDTH	The image width should be a multiple of 8.
VTRANSNM_ERR_INVALID_HEIGHT	The image height should be a multiple of 8.
VTRANSNM_ERR_INTERNAL_ERROR	Contact the vendor. Triggered as assert.

Description

These error codes are either triggered as asserts or passed as **args.errorCode** in the installed errorFunc.

tmolVtransNMControlCommand_t

```
typedef enum tmolVtransNMControlCommand {
   VTRANSNM_CMD_FIELD_INSERTION    = tsaCmdUserBase + 0,
   VTRANSNM_CMD_MEDIAN_FILTER      = tsaCmdUserBase + 1,
   VTRANSNM_CMD_FILM_DETECTION     = tsaCmdUserBase + 2,
   VTRANSNM_CMD_MOTION_ESTIMATION  = tsaCmdUserBase + 3
} tmolVtransNMControlCommand_t;
```

Fields

`VTRANSNM_CMD_FIELD_INSERTION`	Indicates the processing should be field insertion without median filtering. This is the computationally least expensive processing mode available.
`VTRANSNM_CMD_MEDIAN_FILTER`	Indicates the processing should be field insertion with median filtering. This is requires a little more computation than **VTRANSNM_CMD_FIELD_INSERTION** but less than **VTRANSNM_CMD_MOTION_ESTIMATION**.
`VTRANSNM_CMD_FILM_DETECTION`	Indicates the processing should be film input detection plus heuristic deinterlacing. This mode is the default mode. It takes more processing power than median filtering and does a very good job in the deinterlacing. There is no motion compensation.
`VTRANSNM_CMD_MOTION_ESTIMATION`	Indicates the processing should be motion estimated and compensated deinterlacing.

Description

These commands can be passed as 'command' in a **tsaControlArgs_t** structure that is passed to **tmolVtransNMInstanceConfig**. 'parameter' of the **tsaControlArgs_t** structure has no meaning.

VtransNM API Functions

This section presents the VtransNM component functional interface.

tmolVtransNMGetCapabilities

```
extern tmLibappErr_t tmolVtransNMGetCapabilities(
   ptmolVtransNMCapabilities_t   *cap
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function fills in the pointer of a static structure, **tmolVtransNMCapabilities_t**, maintained by the natural motion video transformer, to describe the capabilities and requirements of this library.

tmolVtransNMOpen

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

Parameters

`instance`	Pointer to the (returned) instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation failed.
`TMLIBAPP_ERR_MODULE_IN_USE`	No more instances are available. Currently only one instance is supported, due to the amount of memory and processing power requirements.

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

Description

Opens an instance of the VtransNM component.

The VtransNM task is created with preemption. Usually the task should have low priority. The default stack size is set to 10 kb.

tmolVtransNMInstanceSetup

```
extern tmLibappErr_t tmolVtransNMInstanceSetup(
   Int                              instance,
   ptmolVtransNMInstanceSetup_t     setup
);
```

Parameters

`instance`	Instance previously opened by **tmolVtransNMOpen**.
`setup`	Pointer to the setup data. See **tmolVtransNMInstanceSetup_t**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is not an instance opened with **tmolVtransNMOpen**, triggered via tmAssert.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmolVtransNMOpen**, triggered via tmAssert.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	No memory could be allocated for one of the packets or buffers or internal buffer management structures.
`VTRANSNM_ERR_PASSED_IN_BUFFER_TOO_SMALL`	A non-Null buffer pointer was passed, but the buffer was too small. See **tmolVtransNMConfig_t**. Triggered as assert.
`VTRANSNM_ERR_BUFFER_WIDTH`	Image width should be a multiple of 8.
`VTRANSNM_ERR_BUFFER_HEIGHT`	Image height should be a multiple of 8.

The function can also return any error code produced by **tsaDefaultInstanceSetup**, **tmosTaskCreate**, **tmosSemaphoreCreate**, **tmosTaskStart**, or **tmosTaskSuspend**.

Description

The instance previously opened by **tmolVtransNMOpen** is set up. Memory is allocated for the TSSA packets and video buffers, and to store all the buffer information; **tmolVtransNMInstanceSetup** should be called only once for each instance.

tmolVtransNMGetInstanceSetup

```
extern tmLibappErr_t tmolVtransNMGetInstanceSetup(
   Int                              instance,
   ptmolVtransNMInstanceSetup_t  *setup
);
```

Parameters

`instance`	The instance.
`setup`	Pointer to a variable in which to return a pointer to the setup data.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is not an instance opened with **tmolVtransNMOpen**, triggered via tmAssert.
`TMLIBAPP_ERR_NOT_OPEN`	Instance is not opened with **tmolVtransNMOpen**, triggered via tmAssert.

Description

This function is used during initialization of the transformer. It returns the default settings for the transformer's 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 **tmolVtransNMInstanceSetup**.

tmolVtransNMStart

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

Parameters

`instance`	Instance previously opened by **tmolVtransNMOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is not an instance opened with **tmolVtransNMOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	Instance is not opened with **tmolVtransNMOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	Instance is not set up with **tmolVtransNMInstanceSetup**, triggered via **tmAssert**.

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

Description

The previously opened and set up instance of the transformer is started. Because the transformer creates its own buffers, it is not expected that there are any packets in the input and output queue.

tmolVtransNMStop

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

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is not an instance opened with **tmolVtransNMOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	Instance is not opened with **tmolVtransNMOpen**, triggered via **tmAssert**.
`TMLIBAPP_OK`	Success.

The function can also return any code produced by **tsaDefaultStop**, **tmosSemaphoreP**.

Description

The **tsaDefaultStop** takes care of stopping the instance. The VtransNM instance will do a **tsaDefaultStopPin** on the output component (when that pin was not stopped) and restart it again, to make sure there are no packets left to be displayed. This is done to ensure the reference counts to all buffers are as low as possible. There can be still some buffers in the output chain. When the application wants to reconnect the output queue, the output chain needs to be stopped first (to make sure all buffers are returned to the VtransNM instance such that the reference counts can be set to 0).

After a call to stop, the VtransNM instance can be restarted via a call to Start.

tmolVtransNMClose

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

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is not an instance open with **tmolVtransNMOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_STOPPED`	Instance is not stopped before. Triggered via **tmAssert**.
The function can also return any code produced by **tsaDefaultClose**, **tmosTaskDestroy**, or **tmosSempahoreDestroy**.	

Description

Closes a stopped VtransNM instance. The instance cannot be used anymore. A new instance can be created with **tmolVtransNMOpen**, but all buffers will be allocated again. So when VtransNM allocated the video buffers (as opposed to the application passing in a buffer from which the video buffers are allocated from) these have to be deallocated first to free the memory.

tmolVtransNMInstanceConfig

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

Parameters

`instance`	The instance.
`flags`	Presently ignored.
`args`	**args–>command** is one of the command codes from **tmolVtransNMControlCommand_t**. There are no other required fields to be set in args.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is not an instance open with **tmolVtransNMOpen**. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	Instance is not open. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	Instance is not set up. Triggered via tmAssert.

Description

Switch between different deinterlacing algorithms.

Chapter 11

MPEG Video Decoder (VdecMpeg) 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.

Note
The VdecMpeg is an implementation of the "Recommendation ITU-T H262, ISO/IEC 13818-2" standard.

VdecMpeg API Overview

The VdecMpeg component is a software TSSA MPEG-2 video decoder. It accepts MPEG-1 and MPEG-2 MP@ML video elementary streams. VdecMpeg detects and recovers from bit stream errors but it performs no error concealment. Presentation time stamps (if present) are attached to the outgoing video packets. Decoding time stamps (if present) are compared with an installed reference clock. The result of this comparison is then used by the decoder to determine when the decoding of a video frame must be skipped in order to maintain synchronization with other components. The skipping based on DTS comparison is only done for B-frames.

Normally, VdecMpeg requires 4 output frame buffers. A special "still" mode has been added which allows VdecMpeg to run with 1 output frame buffer. However, in this mode, VdecMpeg is capable of decoding only 1 I-frame before it must be stopped and restarted.

The user can request that user data be extracted from the incoming video stream and passed to a component which resides down stream from the video decoder.

Limitations

VdecMpeg does not run on the TM-1000. The VdecMpeg component uses instructions supported by the TM-1100 and later processors to reduce the processing load. This decoder relies on the TM-1*xxx* family VLD. It will therefore not run on TM-2*xxx* processors.

The decoder is not re-entrant, which means that only one decoder can be alive at any point in time.

VdecMpeg Inputs and Outputs

Overview

The input and outputs of the MPEG video decoder are depicted in Figure 13, following. The data input should be an MPEG video elementary stream, with optional timestamps. The two outputs are; (1) the decoded video stream and (2) a data stream that contains extracted user data. The latter is only sent along on user request. Via the control input, the component can be controlled.

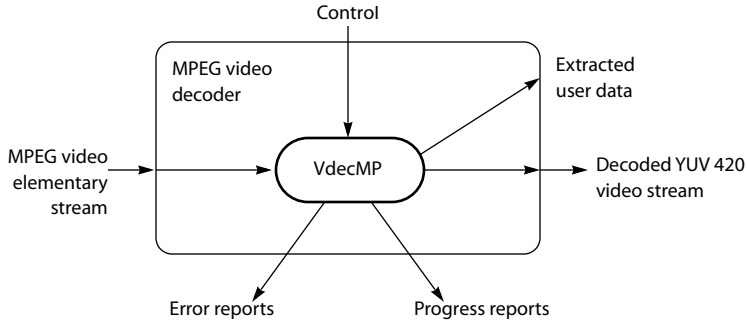


Figure 13 Overview of the Decoder

Inputs

VdecMpeg only operates in data streaming mode. Input packets are requested via the registered datain callback function. Input packets should be of the type tmAvPacket_t. Typically, VdecMpeg retains possession of two input packets. To avoid copying the incoming data, no internal buffering of the input stream is done. Therefore, to ensure efficient operation, the component immediately upstream from VdecMpeg should maintain a rate buffer for the incoming data.

Timestamps are passed in with data packets. The timestamps of packets with the avhValidTimestamp flag set, are used as PTS values, unless also the avhValidDts flag is set, in which case the timestamp is used as DTS value. DTS timestamps extracted from empty packets are associated with the next non-empty input packet. The PTS values are attached to the next decoded video frame and passed to the component immediately down stream from VdecMpeg along with the decoded video frame. If more than one PTS is received for a particular video frame, VdecMpeg always uses the last value received.

The capability format for the input descriptor is set to

```
tmAvFormat_t input_format = {
   sizeof(tmAvFormat_t),              /* size        */
   0,                                 /* hash        */
   0,                                 /* referenceCount */
   avdcVideo,                         /* dataClass   */
   vtfMPEG,                           /* dataType    */
   vmfMPEG1 | vmfMPEG2 | vmfNone,     /* dataSubtype */
   0                                  /* description */
};
```

Outputs

VdecMpeg has two outputs. One output contains the decoded video frames. The other contains user data which has been extracted from the incoming stream.

In the case of video output, each packet contains one entire video frame. In the case of interlaced frames, one tmAvPacket contains both fields. The top field is located at the location indicated by the tmAvPacket's data pointer. While the bottom field is located at data pointer + stride (the regular vdfInterlaced format).

PTS values for the video output are located in the timeStamp field of the tmAvPacket. If the incoming stream contains valid PTS values, the decoder will linearly extrapolate these PTS values such that every decoded video frame out of the decoder will have a PTS. The extrapolated PTS values are only used if the incoming video frame does not have a valid PTS.

The decoded video output has its capability format set to:

```
tmAvFormat_t videoFormat = {
   sizeof(tmAvFormat_t),/* size        */
   0,                   /* hash        */
   0,                   /* referenceCount */
   avdcVideo,           /* dataClass   */
   vtfYUV,              /* dataType    */
   vdfYUV420Planar,     /* dataSubtype */
   0                    /* description */
};
```

The user data output contains user data extracted from the Sequence, GOP and Picture layers of the bitstream. The user can dynamically enable or disable extraction of these user data streams via the command interface. However, the I/O descriptors must be initialized at instance setup. The packets contain un-interpreted data from the bitstream. If the data does not fit in one packet, an error conditions is signalled. Once such an error has occurred, the remaining user data is discarded and a full but incomplete user data packet is sent to the user data output. User data packets are sent out immediately if reordering is not enabled. Otherwise, user data is sent to its output when the corresponding video frame is sent to the video output.

The data output has the following format:

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

The output descriptor assignment is:

```
#define VDECMPEG_OUTPUT        0
#define VDECMPEG_DATA_OUTPUT   1
```


VdecMpeg Errors

VdecMpeg detects and recovers from a wide variety of bitstream errors. Errors are reported via the registered error callback function. Errors which are reported with the **tsaErrorFlagsFatal** set should result in termination of the instance. Bitstream errors are never fatal. It is assumed that the incoming data stream is a stream that the decoder should be able to decode.

Once an error has been reported, the default recovery mechanism is to seek to the next group of pictures and resume decoding at that point. Video frames in which an error was encountered part way through the decoding process are sent to the down stream component with buffersInUse set to 0. Note that this is the only time which it is acceptable to return buffers to the video decoder out of order. In all other instances, video frame buffers must be returned to the video decoder in the order in which they were sent.

VdecMpeg Progress

There are three progress reports produced by VdecMpeg. The decoder reports the decode of a frame (and frame type), the skip of a frame and the sequence information from which the application can determine what type of bitstream is decoded. The VdecMpeg component uses the tsaProgressFlagChangeFormat, which is handled by TSSA internally, to install a format on the Video output queue.

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

VdecMpeg Configuration

The following control modes can be set via calls to tmolVdecMpegInstanceConfig:

1. Enable extraction of user data. Sending this command will tell the video decoder to extract user data and send it to the data output. The argument to this command is a boolean which indicates whether or not to reorder the user data with the decoded video frames. A value of false causes the user data to be sent down stream immediately. Note that there are actually 3 separate commands. One each to enable sequence, GOP and picture user data independently.
2. Disable extraction of user data. Sending this command disables extraction of user data by the video decoder. Again, there are actually 3 separate commands. One each to enable sequence, GOP and picture user data independently.
3. Flush. Assumed to be called only when there are no more input packets, In this case a flush buffer is installed in the VLD, the last data is decoded and then the decoded frames, if any, are sent out.
4. Ignore DTS, in which case all incoming frames are decoded regardless of their decoding time stamp. This mode can be used to implement trick modes.

5. Resume decoding with taking DTS into account, the default operation mode when a clock is installed.

VdecMpeg API Data Structures

This section describes the VdecMpeg component data structures.

tmolVdecMpegInstanceSetup_t, tmalVdecMpegInstanceSetup_t

```
typedef struct tmalVdecMpegInstance {
   ptsaDefaultInstanceSetup_t      defaultSetup;
   UInt32                          imageStride;
   UInt32                          numberOfOutputPackets;
   UInt32                          numberOfUserDataPackets;
} tmalVdecMpegInstanceSetup_t, *ptmalVdecMpegInstanceSetup_t;

typedef tmalVdecMpegInstanceSetup_t tmolVdecMpegInstanceSetup_t;
typedef ptmalVdecMpegInstanceSetup_t ptmolVdecMpegInstanceSetup_t;
```

Fields

`defaultSetup`	See TSSA documentation
`imageStride`	In case the display component that interprets the video output has a stride restriction, for instance the ICP. When set to 0, no stride restriction is assumed and the image width of the decoder picture is taken as stride.
`numberOfOutputPackets`	Either set to 1 or 4. When set to 1, only one I frame is decoded.
`numberOfUserDataPackets`	Total number of packets available for user data.

Description

Data structure passed to **tmolVdecMpegInstanceSetup** or **tmalVdegMpegInstanceSetup** to describe the input and output connections and other initial values.

tmolVdecMpegCapabilities_t, tmalVdecMpegCapabilities_t

```
typedef struct tmalVdecMpegCapabilities{
   ptsaDefaultCapabilities_t   defaultCaps;
} tmalVdecMpegCapabilities_t, *ptmalVdecMpegCapabilities_t;

typedef tmalVdecMpegCapabilities_t tmolVdecMpegCapabilities_t;
typedef ptmalVdecMpegCapabilities_t ptmolVdecMpegCapabilities_t;
```

Fields

`defaultCaps`	See TSSA documentation.

Description

For input and output descriptors, see *VdecMpeg Inputs and Outputs* on page 178. The text section of VdecMpeg is about 100 kb and the initialized data section is about 4 kb. There is no bss requirement.

tmolVdecMpegErrorFlags_t

Err_base_VdecMpeg is 0x13070000.

```
typedef enum {
/* Fatal errors */
   VDECMPEG_ERR_VLD_OPEN_FAILED            Err_base_VDecMpeg + 0x0001,
   VDECMPEG_ERR_INVALID_PROCESSOR          Err_base_VdecMpeg + 0x0002,
   VDECMPEG_ERR_NO_PICTURE_INFO_ALLOCATED  Err_base_VdecMpeg + 0x0003,
/* Non-fatal errors (action by decoder itself) */
   VDECMPEG_ERR_RESERVED_EXT_STARTCODE_ID  Err_base_VdecMpeg + 0x0100,
   VDECMPEG_ERR_UNEXPECTED_STARTCODE       Err_base_VdecMpeg + 0x0101,
   VDECMPEG_ERR_ODD_FIELD_PICTURES         Err_base_VdecMpeg + 0x0102,
   VDECMPEG_ERR_LAST_FRAME_NOT_COMPLETE    Err_base_VdecMpeg + 0x0103,
   VDECMPEG_VLD_ERROR                      Err_base_VdecMpeg + 0x0104,
   VDECMPEG_ERR_MBA_OVERFLOW               Err_base_VdecMpeg + 0x0105,
   VDECMPEG_ERR_MBA_EXCEEDS_PICTURE_SIZE   Err_base_VdecMpeg + 0x0106,
   VDECMPEG_ERR_DCT_COEFFS_EXCEED_64       Err_base_VdecMpeg + 0x0107,
   VDECMPEG_ERR_INVALID_MOTION_TYPE        Err_base_VdecMpeg + 0x0108,
   VDECMPEG_ERR_AV_BUFFERS_TOO_SMALL       Err_base_VdecMpeg + 0x0109,
   VDECMPEG_ERR_ONLY_420_SUPPORTED         Err_base_VdecMpeg + 0x010A,
   VDECMPEG_ERR_ONLY_MPML_SUPPORTED        Err_base_VdecMpeg + 0x010B,
   VDECMPEG_ERR_SPATIAL_SCALABILITY_NOT_SUPPORTED
                                           Err_base_VdecMpeg + 0x010C,
   VDECMPEG_ERR_TEMPORAL_SCALABILITY_NOT_SUPPORTED
                                           Err_base_VdecMpeg + 0x010D,
   VDECMPEG_ERR_INTERNAL_ERROR             Err_base_VdecMpeg + 0x01FF
} tmalVdecMpegErrorFlags_t;
```

Fields

Fatal errors

`VDECMPEG_ERR_VLD_OPEN_FAILED` The VLD open failed, the interrupt could not be allocated.

`VDECMPEG_ERR_INVALID_PROCESSOR` The decoder is executed on a TM-1000 processor. For speed and compliance reasons, some special instruction supported by TM-1100 or later processors are required.

`VDECMPEG_ERR_NO_PICTURE_INFO_ALLOCATED` A video packet was taken from the queue and it did not have a preallocted **tmalVdecMpegPictureInfo_t** installed in the userPointer.

`VDECMPEG_ERR_RESERVED_EXT_STARTCODE_ID` An unknown extension start code was encountered. The extension startcode id is returned in the args.description field. Decoding resumes at the next start code.

`VDECMPEG_ERR_UNEXPECTED_STARTCODE`
: A non-video startcode was encountered. The startcode is described in the args.description field. Decoding is restarted at the next GOP.

`VDECMPEG_ERR_ODD_FIELD_PICTURES`
: An odd number of field pictures has been encountered before the current frame picture.

`VDECMPEG_ERR_LAST_FRAME_NOT_COMPLETE`
: An odd number of field pictures was decoded before a sequence end code was encountered.

`VDECMPEG_VLD_ERROR`
: The VLD has detected and illegal Huffman code.

`VDECMPEG_ERR_MBA_OVERFLOW`
: A macroblock address increment value has exceeded the maximum alllowable value (i.e. number of macroblocks per row). Only valid for MPEG-2 sequences.

`VDECMPEG_ERR_MBA_EXCEEDS_PICTURE_SIZE`
: The number of macrblocks decoded for the current picture has exceeded the picture size specified in the sequence header.

`VDECMPEG_ERR_DCT_COEFFS_EXCEED_64`
: A block with more than 64 DCT coefficients has been encountered.

`VDECMPEG_ERR_INVALID_MOTION_TYPE`
: The motion type for the current macroblock is illegal with respect to the current picture structure.

`VDECMPEG_ERR_AV_BUFFERS_TOO_SMALL`
: The given YUV output buffers were to small to decode this bitstream. Decoding is restarted at the next gop. The user may want to stop the instance, insert bigger buffers and restart.

`VDECMPEG_ERR_ONLY_420_SUPPORTED`
: A chroma format value (see 13818-2) other than 1 has been encountered in the sequence extension. For MP@ML streams, the chroma format field can only be 1. Decoding is restarted at the next gop.

`VDECMPEG_ERR_ONLY_MPML_SUPPORTED`
: Only main profile, main level is supported, decoding isrestarted at the next gop.

`VDECMPEG_ERR_SPATIAL_SCALABILITY_NOT_SUPPORTED`
: A spatial scalable extension (picture or sequence) has been detected. No such extensions are allowed in MP@ML streams. Decoding is restarted at the next GOP.

`VDECMPEG_ERR_TEMPORAL_SCALABILITY_NOT_SUPPORTED`
: A temporal scalable extension (picture or sequence) has been detected. No such extensions

	are allowed in MP@ML streams. Decoding is restarted at the next gop.
VDECMPEG_ERR_INTERNAL_ERROR	Contact the vendor, an internal error has occurred.

Description

These error codes are passed as **args.errorCode** in the installed errorFunc. Only when explicitly mentioned the description field is set. Usually the **args.description** is set to Null.

tmalVdecMpegControlCommand_t

```
typedef enum {
   VDECMPEG_CMD_FREEZE        tsaCmdUserBase +  Ø,
   VDECMPEG_CMD_UNFREEZE      tsaCmdUserBase +  1,
   VDECMPEG_CMD_IGNORE_DTS    tsaCmdUserBase +  2,
   VDECMPEG_CMD_NORMAL_DTS    tsaCmdUserBase +  3,
   VDECMPEG_CMD_SEQ_UD_ON     tsaCmdUserBase +  4,
   VDECMPEG_CMD_SEQ_UD_OFF    tsaCmdUserBase +  5,
   VDECMPEG_CMD_GOP_UD_ON     tsaCmdUserBase +  6,
   VDECMPEG_CMD_GOP_UD_OFF    tsaCmdUserBase +  7,
   VDECMPEG_CMD_PIC_UD_ON     tsaCmdUserBase +  8,
   VDECMPEG_CMD_PIC_UD_OFF    tsaCmdUserBase +  9,
   VDECMPEG_CMD_FLUSH         tsaCmdUserBase + 1Ø
   VDECMPEG_CMD_SKIP_BFRAMES  tsaCmdUserBase + 11
} tmalVdecMpegControlCommand_t;
```

Fields

`VDECMPEG_CMD_FREEZE`	CURRENTLY UNIMPLEMENTED. Indicates the output picture needs to be frozen. The decoder will decode I and P frames (if the number of buffers set by instance setup allows this), such that unfreeze is smooth and quick. When the decoder was frozen this command has no effect.
`VDECMPEG_CMD_UNFREEZE`	CURRENTLY UNIMPLEMENTED. Unfreezes a frozen decoder. When the decoder is not frozen this command has no effect.
`VDECMPEG_CMD_IGNORE_DTS`	Decodes all incoming frames regardless of whether the DTS has expired.
`VDECMPEG_CMD_NORMAL_DTS`	Interpret the DTS, when the DTS has expired, do not decode the frame. This is the default operation mode when a valid clock is passed in the instance setup.
`VDECMPEG_CMD_SEQ_UD_ON`	Enable extraction of user data at the sequence level. A boolean cast of "parameter" is used to indicate whether the extracted user data should be reorderd with the outgoing video frames.
`VDECMPEG_CMD_SEQ_UD_OFF`	Disable extraction of user data at the sequence level.
`VDECMPEG_CMD_GOP_UD_ON`	Enable extraction of user data at the group of pictures level. A boolean cast of "parameter" is used to indicate whether the extracted user data should be reorderd with the outgoing video frames.

VDECMPEG_CMD_GOP_UD_OFF	Disable extraction of user data at the group of pictures level.
VDECMPEG_CMD_PIC_UD_ON	Enable extraction of user data at the picture level. A boolean cast of "parameter" is used to indicate whether the extracted user data should be reorderd with the outgoing video frames.
VDECMPEG_CMD_PIC_UD_OFF	Disable extraction of user data at the picture level.
VDECMPEG_CMD_FLUSH	Decode all data that has been passed to the decoder. Flush the decoded output pictures. It is assumed that there is no incoming data anymore.
VDECMPEG_CMD_SKIP_BFRAMES	Skip decoding of B-frames.

Description

These commands can be passed as 'command' in a **ptsaControlArgs_t** structure that is passed to **tmolVdecMpegInstanceConfig**. Unless otherwise indicated, 'parameter' of the **ptsaControlArgs_t** structure has no meaning.

tmalVdecMpegProgressFlags_t

```
typedef enum {
   VDECMPEG_NEW_SEQUENCE    = 0x0001,
   VDECMPEG_DECODED_A_FRAME = 0x0002,
   VDECMPEG_SKIPPED_A_FRAME = 0x0004,
   VDECMPEG_TIMEDIFF        = 0x0008
} tmalVdecMpegProgressFlags_t;
```

Fields

VDECMPEG_NEW_SEQUENCE	A new sequence header is encountered, see **tmalVdecMpegSequenceDescription_t**.
VDECMPEG_DECODED_A_FRAME	A frame was successfully decoded. The **args.description** field is set to I_TYPE, P_TYPE, or B_TYPE and indicates which frame has just been decoded. I_TYPE etc are defined in the tmalVdecMpeg.h include file.
VDECMPEG_SKIPPED_A_FRAME	A frame was skipped because the DTS was expired. The args.description field is set to B_TYPE since these are the only type of frames the decoder can safely skip.
VDECMPEG_TIMEDIFF	Reserved for future use.

Description

Used in progress reports, as **args.progressCode** in the **ptsaProgressArgs_t** structure.

tmalVdecMpegSequenceLevel_t

```
typedef enum {
   VDECMPEG_MPEG1_SEQ,
   VDECMPEG_MPEG2_SEQ
} tmalVdecMpegSequenceLevel_t;
```

Fields

VDECMPEG_MPEG1_SEQ	Indication of MPEG level 1 sequence.
VDECMPEG_MPEG2_SEQ	Indication of MPEG level 2 sequence.

Description

This data structure is used in **VDECMPEG_NEW_SEQUENCE** progress report. It is passed via the **tmalVdecMpegSequenceDescription_t** structure.

tmalVdecMpegSequenceDescription_t

```
typedef struct{
   UInt32                       size;
   tmalVdecMpegSequenceLevel_t  level;
   UInt32                       imageWidth;
   UInt32                       imageHeight;
   UInt32                       bitRateValue;
   UInt16                       aspectRatio;
   Bool                         progressiveSequence;
   Bool                         sequenceDisplayExtensionPresent;
} tmalVdecMpegSequenceDescription_t, *ptmalVdecMpegSequenceDescription_t;
```

Fields

`size`	Used by TSSA, always the size of the structure.
`level`	Either **VDECMPEG_MPEG1_SEQ** or **VDECMPEG_MPEG2_SEQ**.
`imageWidth`	The width of the decoded fields as indicated in the sequence header.
`imageHeight`	The height of the decoded fields as indicated in the sequence header.
`bitRateValue`	The bit rate as indicated in the sequence header.
`aspectRatio`	The aspect ratio as indicated in the sequence header.
`progressiveSequence`	Whether this bitstream is progressive or interlaced.
`sequenceDisplayExtensionPresent`	Whether this sequence has display extension set.

Description

This data structure is passed by reference in the description field of the **ptsaProgressArgs_t** structure that is passed to the installed progress function. The **progressCode** is set to **VDECMPEG_NEW_SEQUENCE**.

tmalVdecMpegPictureInfo_t

```
typedef struct{
   UInt32       size;
   UInt32       dataFormat;
   Int16        displayHorizontalSize;
   Int16        displayVerticalSize;
   Int16        frameCentreHorizOffset[3];
   Int16        frameCentreVertOffset[3];
   UInt16       aspectRatio;
   ptmAvFormat  userData[MAX_UD_INDEX];
} tmalVdecMpegPictureInfo_t, *ptmalVdecMpegPictureInfo_t;
```

Fields

`size` — Used by TSSA, the size of this structure.

`dataFormat` — Data format, defined as follows:

`dataFormat =`

```
   ((picture_structure    & VO_DF_PS_MASK         ) << VO_DF_PS_SHIFT)         |
   ((chroma_format        & VO_DF_CF_MASK         ) << VO_DF_CF_SHIFT)         |
   ((matrix_coefficients  & VO_DF_COL_CONV_MASK   ) << VO_DF_COL_CONV_SHIFT)   |
   ((progressive_frame    & VO_DF_PROG_FR_MASK    ) << VO_DF_PROG_FR_SHIFT)    |
   ((top_field_first      & VO_DF_TFF_MASK        ) << VO_DF_TFF_SHIFT)        |
   ((repeat_first_field   & VO_DF_RFF_MASK        ) << VO_DF_RFF_SHIFT)        |
   ((progressive_sequence & VO_DF_PROGSEQ_MASK    ) << VO_DF_PROGSEQ_SHIFT)    |
   (((picture_rate-1)     & VO_DF_FRAME_RATE_MASK) << VO_DF_FRAME_RATE_SHIFT)|
   ((pict_type)           & VO_DF_PTYPE_SHIFT     );
```

picture_structure

TOP_FIELD	0x1	Frame is encoded in the form of 2 fields and current field is the top field.
BOTTOM_FIELD	0x2	Frame is encoded in the form of 2 fields and current field is the bottom field.
FRAME_PICTURE	0x3	Both fields are encoded as one single frame. This is also the case for MPEG-1 encoded streams.

chroma_format

This 2 bit integer indicates the chrominance format. For VdecMP, only **CHROMA420** is supported.

CHROMA420 0x1	4:2:0 format.
CHROMA422 0x2	4:2:2 format.
CHROMA444 0x3	4:4:4 format.

matrix_coefficients

This 8 bits integer describes the matrix coefficients used to perform RGB to YCrCb conversion. In the case there is no **sequence_display_extension** in the bit stream, the matrix coefficients is determined by the recommendation ITU_R BT.709.

progressive_frame

When set to zero, it indicates that the 2 fields of the frame are interlaced fields. When set to 1, it indicates that the 2 fields of the frame are from the same time instant as one another.

progressive_sequence

When set to 1, the video sequence contains only progressive frame-pictures (for instance as in MPEG-1), when set to 0, video sequence can contain both frame-picture and field-picture, and frame-pictures may be interlaced or progressive.

top_field_first

If **progressive_sequence == 0**, top_field_first set to 1 indicates that the top field of the reconstructed frame is the first field output by the decoding process. If **progressive_sequence == 1**, this field, combined with **repeat_first_field** indicates how many times the reconstructed frame is output by the decoding process.

repeat_first_field

This flag is applicable only in a frame picture. In case **progressive_frame == 1**, and **progressive_sequence == 0**, if set to 1, then the first field is displayed, then the other field, and then the first field is repeated.

pict_type

The picture coding type. Not used by any renderer.

All this bit stream information is packed into one 32-bit **dataFormat** register, as defined previously, using the following masks:

`VO_DF_PS_MASK`	0x3 2 bits mask.
`VO_DF_CF_MASK`	0x3 2 bits mask.
`VO_DF_COL_CONV_MASK`	0x7 3 bits mask.
`VO_DF_PROG_FR_MASK`	0x1 1 bit mask.
`VO_DF_TFF_MASK`	0x1 1 bit mask.
`VO_DF_RFF_MASK`	0x1 1 bit mask.
`VO_DF_PROGSEQ_MASK`	0x1 1 bit mask.
`VO_DF_LASTFRAME_MASK`	0x1 1 bit mask.
`VO_DF_FRAME_RATE_MASK`	0xF 4 bits mask.
`VO_DF_FRAME_SENT_MASK`	0x1 1 bit mask (for internal use).
`VO_DF_PTYPE_MASK`	0x3 2 bits mask.

VO_DF_PS_SHIFT	0x0
VO_DF_CF_SHIFT	0x2
VO_DF_COL_CONV_SHIFT	0x4
VO_DF_PROG_FR_SHIFT	0x7
VO_DF_TFF_SHIFT	0x8
VO_DF_RFF_SHIFT	0x9
VO_DF_PROGSEQ_SHIFT	0xA
VO_DF_LASTFRAME_SHIFT	0xB
VO_DF_FRAME_RATE_SHIFT	0xC
VO_DF_FRAME_SENT_MASK	0x10 (for internal use)
VO_DF_PTYPE_SHIFT	0x11

Fields, *continued*

displayHorizontalSize	
displayVerticalSize	These two fields define a display rectangle considered as the intended display area. If it is smaller than the encoded frame size, then only a portion of the encoded frame is displayed.
frameCentreHorizOffset	
frameCentreVertOffset	These two fields indicate the position of the center of the display rectangle. If both are 0, the center of the display rectangle is located at the center of the decoded frame. Those 2 fields are in 1/16th sample units.
aspectRatio	This field gives the display aspect ratio: 3/4, 16/9 or 1/2.21.
userData	Contains three pointers to memory buffers where the user data extracted by the decoder will be stored.

Description

This data structure is passed via the **userPointer** field of the **tmAvHeader_t** of each video packet sent out. The **format.description** field has the **vdfMPEGExtension** flag set, which indicates to the renderer that the packet has an MPEG extension attached to it.

VdecMpeg API Functions

This section presents the VdecMpeg component functional interface.

tmolVdecMpegGetCapabilities, tmalVdecMpegGetCapabilities

```
extern tmLibappErr_t tmolVdecMpegGetCapabilities(
   ptmolVdecMpegCapabilities_t    *cap
);
extern tmLibappErr_t tmalVdecMpegGetCapabilities(
   ptmolVdecMpegCapabilities_t    *cap
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.

Description

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

tmolVdecMpegOpen, tmalVdecMpegOpen

```
extern tmLibappErr_t tmolVdecMpegOpen(
   Int   *instance
);
extern tmLibappErr_t tmalVdecMpegOpen(
   Int   *instance
);
```

Parameters

`instance`	Pointer to the (returned) instance.

Return Codes

`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation failed.
`TMLIBAPP_ERR_MODULE_IN_USE`	No more instances are available. Currently only one instance is supported, due to the amount of memory and processing power needed.
`VDECMPEG_ERR_INVALID_PROCESSOR`	Attempt to run the decoder on a TM-1000. It needs a TM-1100 (or later) processor.
`TMLIBAPP_OK`	Success.

Or, in case of **tmolVdecMpegOpen**, any return code produced by **tsaDefaultOpen**.

Description

Opens an instance of the VdecMpeg component.

The VdecMpeg task is created with preemption. Usually the task should have low priority. The default stack size is set to 10 kb.

tmolVdecMpegInstanceSetup, tmalVdegMpegInstanceSetup

```
extern tmLibappErr_t tmolVdecMpegInstanceSetup(
   Int                           instance,
   ptmolVdecMpegInstanceSetup_t setup
);
extern tmLibappErr_t tmalVdecMpegInstanceSetup(
   Int                           instance,
   ptmolVdecMpegInstanceSetup_t setup
);
```

Parameters

`instance`	Instance previously opened by **tmolVdecMpegOpen**, **tmalVdecMpegOpen**.
`setup`	Pointer to the demultiplexer's setup data structure, see **tmolVdecMpegInstanceSetup_t, tmalVdecMpegInstanceSetup_t**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolVdecMpegOpen, tmalVdecMpegOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmolVdecMpegOpen, tmalVdecMpegOpen**, triggered via **tmAssert**.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	No memory could be allocated for the instance.
`TMLIBAPP_ERR_INVALID_SETUP`	When the numbers of output buffers is not 1 or 4. See *Limitations* on page 157.

The function **tmolVdecmpegInstanceSetup** can return any code produced by **tsaDefaultInstanceSetup**.

Description

The instance previously opened by **tmolVdecMpegOpen** is set up. Memory is allocated for the internally held buffers that are needed for decoding. **tmolVdecMpegInstanceSetup** should be called only once for each instance.

tmolVdecMpegGetInstanceSetup, tmalVdecMpegGetInstanceSetup

```
extern tmLibappErr_t tmolVdecMpegInstanceSetup(
   Int                            instance,
   ptmolVdecMpegInstanceSetup_t *setup
);
extern tmLibappErr_t tmalVdecMpegInstanceSetup(
   Int                            instance,
   ptmolVdecMpegInstanceSetup_t *setup
);
```

Parameters

`instance`	The instance.
`setup`	Pointer to a variable in which to return a pointer to the setup data.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolVdecMpegOpen, tmalVdecMpegOpen**. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmolVdecMpegOpen, tmalVdecMpegOpen**. Triggered via **tmAssert**.

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 **tmolVdecMpegInstanceSetup** or **tmalVdegMpegInstanceSetup**.

tmolVdecMpegStart, tmalVdecMpegStart

```
extern tmLibappErr_t tmolVdecMpegStart(
   Int   instance
);
extern tmLibappErr_t tmalVdecMpegStart(
   Int   instance
);
```

Parameters

`instance`	Instance previously opened by **tmolVdecMpegOpen** or **tmalVdecMpegOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolVdecMpegOpen, tmalVdecMpegOpen**.Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmolVdecMpegOpen, tmalVdecMpegOpen**. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	When the instance is not set up with **tmolVdecMpegInstanceSetup, tmalVdegMpegInstanceSetup**. Triggered via **tmAssert**.

The function **tmolVdecMpegStart** can return any code produced by **tsaDefaultStart**.

Description

The previously opened and set up instance of the decoder is started. It is expected that the empty queues of the video output contains empty video packets, with allocated **tmalVdecMpegPictureInfo_t** allocated and assigned to the **userPointer** of the packets. Then the decoder starts to wait for input data from the input queue.

tmolVdecMpegStop, tmalVdecMpegStop

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

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolVdecMpegOpen, tmalVdecMpegOpen**. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmolVdecMpegOpen, tmalVdecMpegOpen**. Triggered via **tmAssert**.

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

Description

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

tmolVdecMpegClose, tmalVdecMpegClose

```
extern tmLibappErr_t tmolVdecMpegClose(
   Int   instance
);
extern tmLibappErr_t tmalVdecMpegClose(
   Int   instance
);
```

Parameters

instance	The instance.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	When the instance is not a valid instance open with **tmolVdecMpegOpen, tmalVdecMpegOpen**. Triggered via **tmAssert**.
TMLIBAPP_ERR_NOT_STOPPED	When the instance is not stopped before closing. Triggered via **tmAssert**.

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

Description

Closes a stopped VdecMpeg instance.

tmolVdecMpegInstanceConfig

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

Parameters

`instance`	The instance.
`flags`	Presently ignored.
`args`	**args–>command** is one of the command codes from **tmalVdecMpegControlCommand_t**. There are no other required fields to be set in args.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmolVdecMpegOpen**. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmolVdecMpegOpen**. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	When the instance is not set up with **tmolVdecMpegInstanceSetup**. Triggered via **tmAssert**.

Description

See **tmalVdecMpegControlCommand_t** for possible control commands.

tmalVdecMpegInstanceConfig

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

Parameters

`instance`	The instance.
`args`	**args–>command** is one of the command codes from **tmalVdecMpegControlCommand_t**. There are no other required fields to be set in args.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	When the instance is not a valid instance open with **tmalVdecMpegInstanceConfig**. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_OPEN`	When the instance is not opened with **tmalVdecMpegOpen**. Triggered via **tmAssert**.
`TMLIBAPP_ERR_NOT_SETUP`	When the instance is not set up with **tmalVdecMpegInstanceSetup**. Triggered via **tmAssert**.

Description

See **tmalVdecMpegControlCommand_t** for possible control commands. Control commands are handled on all blocking datain and dataout functions.

