Book 7—Video Support Libraries

Part A:

Video I/O

TriMedia

Version 2.0 beta

Table of Contents

Chapter 1

Video In/Out (vi/vo) API

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

Video In/Out API Overview

The TriMedia Video device library provides a set of functions for accessing the Video-in and Video-out peripherals available on TriMedia processors. The TriMedia Video device library controls the Video-in and Video-out hardware on the TriMedia, providing specific functions for controlling video encoders and decoders. It is relinkable with other programs, giving you total control of the hardware. It allows you to:

- Optimize Interrupt Service Routines (ISR) in order to meet application requirements.
- Create vendor-specific initialization and configuration routines for on-board chips, such as a decoder that works with TriMedia Video-in and an encoder that works with the TriMedia Video-out component.

The example applications show how the Video 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 Video In/Out API should be very helpful.

Introduction

The Video In/Out (VI/VO) peripheral provides a digitized stream of video or data into or out of SDRAM. VI and VO have two operation modes:

- Video Stream Input, in which VI/VO interfaces with the decoder/encoder on the board.
- Data Streaming, in which VI/VO interfaces with decoder/encoder, without data selection or data interpretation.

The Enhanced Video Out unit (EVO) is not part of the TriMedia processor. EVO features are controlled through MMIO (**EVO_*xxx***) registers, and the **EVO_ENABLE** bit in the **EVO_CTL** register.

viOpen and voOpen

VI/VO starts by claiming a device using one of the **viOpen**, **viOpenM**, **voOpen**, or **voOpenM** functions.

Instance Setup

Initialization then starts with a call to **viInstanceSetup** or **voInstanceSetup** that sets up the basic functionality and initializes the specified decoder/encoder. It sets the common fields between the two modes of operation. After that, you can choose from:

- **viYUVSetup** and **voYUVSetup** for the video modes operation.
- **viRawSetup** and **voRawSetup** for the data streaming modes of operation.

Changing Buffers

A special interface offers fast buffer switching. It is implemented with the macros:

- **viYUVChangeBuffer/voYUVChangeBuffer(instance, Y, U, V)**
- **viRawChangeBuffer1/voRawChangeBuffer1(instance, B)**,
- **viRawChangeBuffer2/voRawChangeBuffer2(instance, B)**

The Rest

The functions **viStart/voStart** and **viStop/voStop** start and stop the device.

The functions **viClose/voClose** close the device and free memory allocated for the instance.

Some decoder control functions are available:

```
viSetBrightness   viSetContrast   viDetectColorStandard
viSetSaturation   viSetHue
```

Several decoder functions support VBI data processing, internal scaling, and other features.

Caveats

There are a number of hardware bugs in the earlier versions of the TM-1*xxx* chips. Most of them have been addressed. Please read the errata carefully before operating VI/VO.

Error Codes

The error codes returned by the functions of the Video In/Out API are defined in the tmLibdevErr.h.

Video-In API Data Structures

This section presents the Video-in data structures.

Name	Page
viYUVModes_t	12
viRawModes_t	12
viCapabilities_t	13
viRawSetup_t	14
viYUVSetup_t	15
viInstanceSetup_t	16

viYUVModes_t

```
typedef enum {
   viFULLRES = 0,
   viHALFRES = 1,
} viYUVModes_t;
```

Description

Enumerates the possible video-in modes of the VI peripheral. These are used when VI interfaces to a digital or analog camera. Refer to the appropriate TriMedia data book for more information.

viRawModes_t

```
typedef enum {
   viSTREAM8   = 2,
   viSTREAM10S = 3,
   viSTREAM10U = 4,
   viMESSAGE   = 5,
} viRawModes_t;
```

Description

Enumerates the data streaming modes that VI can be set in. These modes are used when VI interfaces with an A/D raw input channel. Refer to the appropriate TriMedia data book for more information.

viCapabilities_t

```
typedef struct {
   tmVersion_t   version;
   Int           numSupportedInstances;
   Int           numCurrentInstances;
   char          codecName[16];
   UInt32        videoStandards;
   UInt32        adapterTypes;
} viCapabilities_t, *pviCapabilities_t;
```

Fields

`version`	Version of the video in library component.
`numSupportedInstances`	Number of instances that are supported by the processor.
`numCurrentInstances`	Number of instances currently in use.
`codecName[16]`	Name of the video encoder on the board as returned by the board init routine.
`videoStandards`	OR'd bitmask of video standards supported by the encoder on the board.
`adapterTypes`	OR'd bitmask of video standards supported by the encoder on the board.

Description

The capabilities of the VI library component can be investigated using **viGetCapabilities** or **viGetCapabilitiesM**. These return a pointer to a read-only data structure of the type **viCapabilities_t** as described here.

viRawSetup_t

```
typedef struct {
   Bool          buf1fullEnable;
   Bool          buf2fullEnable;
   Bool          overflowEnable;
   Bool          overrunEnable;
   viRawModes_t  mode;
   UInt          size;
   Pointer       base1,base2;
} viRawSetup_t, *pviRawSetup_t;
```

Fields

`buf1fullEnable`	Enables the interrupt when buffer 1 is full.
`buf2fullEnable`	Enables the interrupt when buffer 2 is full.
`overflowEnable`	Used in message passing mode.
`overrunEnable`	Used in raw mode.
`mode`	The data streaming mode in which VI has to operate.
`size`	Size of buffers, in bytes.
`base1`, `base2`	Pointers to buffers.

Description

viRawSetup_t is used when VI is used in data streaming mode. This structure should be used with the **viRawSetup** function.

viYUVSetup_t

```
typedef struct {
   Bool          thresholdReachedEnable;
   Bool          captureCompleteEnable;
   Bool          cositedSampling;
   viYUVModes_t  mode;
   UInt          yThreshold;
   UInt          startX, startY;
   UInt          width;
   UInt          height;
   Pointer       yBase,  uBase,  vBase;
   UInt          yDelta, uDelta, vDelta;
} viYUVSetup_t, *pviYUVSetup_t;
```

Fields

`thresholdReachedEnable`	Enable interrupt whenever the threshold is reached.
`captureCompleteEnable`	Enable interrupt when capture is completed.
`cositedSampling`	Co-sited sampling, as opposed to interspersed sampling; refer to the appropriate TriMedia data book for more information.
`mode`	The data streaming mode in which VI has to operate.
`yThreshold`	The line where the threshold interrupt should be generated; refer to the appropriate TriMedia data book for more information.
`startX`	Defines the starting pixel number or x-coordinate for sampling; this must be an even number.
`startY`	Defines the starting pixel number or y-coordinate for sampling.
`width`	Defines the width of the captured image; this must be an even number.
`height`	Defines the height of the captured image.
`yBase`, `uBase`, `vBase`	Pointers to variables in which the captured data is to be stored.
`yDelta`, `uDelta`, `vDelta`	The address differences between the last sample of a line and the address of the first sample of the next line; all deltas should be chosen such that the line start addresses are 64-byte aligned.

Description

This structure should be used with the **viYUVInstanceSetup** function.

viInstanceSetup_t

```
typedef struct {
   Bool                       hbeEnable;
   intPriority_t              interruptPriority;
   void                      (*isr)(void);
   tmVideoAnalogStandard_t    videoStandard;
   tmVideoAnalogAdapter_t     adapterType;
} viInstanceSetup_t, *pviInstanceSetup_t;
```

Fields

`hbeEnable`	Enables highway bandwidth errors.
`interruptPriority`	VI interrupt priority.
`isr`	Pointer to the interrupt service routine.
`videoStandard`	Default standard: the decoder will try to locate the standard but if that fails, this is the default.
`adapterType`	Uses **vaaCVBS** or **vaaSvideo** adapters.

Description

This can be used as the common initialization structure among all video-in modes of operation, including image and data streaming modes. It should be passed to **viInstanc-eSetup**, which performs the initial programming of the video-in peripheral. After this generic setup, use either **viYUVSetup** or **viRawSetup** to complete the initialization.

Video-In API Functions

This section describes the TriMedia Video-in API functions.

Name	Page
viGetNumberOfUnits	20
viGetCapabilities	21
viGetCapabilitiesM	22
viInstanceSetup	23
viYUVSetup	24
viRawSetup	25
viOpen	26
viOpenM	27
viClose	28
viStart	29
viStop	30
viYUVChangeBuffer	31
viRawChangeBuffer1	32
viRawChangeBuffer2	33
viConfigureDecoder	34
viGetColorStandard	35
viSetBrightness	36
viSetContrast	37
viSetHue	38
viSetSaturation	39
viGetVideoStandard	40
viGetVSyncFallingEdge	41
viGetSlicedData	42
viGetStatus	43
viGetSupportedDataServices	44
viSetDataServices	45

Name	Page
viGetSlicerLineFlags	46
viEnableSlicing	47
viSetSlicerVideoStandard	48
viGetSlicerVideoStandard	49
viToggleFieldID	50
viSetSlicerInput	51
viGetSlicerInput	52
viSetVideoColor	53
viGetVideoColor	54
viSetAnalogInput	55
viGetAnalogInput	56
viSetStandard	57
viSetSourceType	58
viGetSourceType	59
viSetOutputFormat	60
viGetOutputFormat	61
viSetAcquisitionWnd	62
viGetAcquisitionWnd	63
viGetDefaultAcquisitionWnd	64
viSetOutputSize	65
viSetInterlaceMode	66
viDisableDecoder	67
viEnablePowerSaveMode	68
viGetGPIOCount	69
viSetGPIOState	70
viGetGPIOState	71
viOpenVBI	72
viEnableVBI	73

Name	Page
viSetVBIMode	74
viSetSlicerMode	75
viCloseVBI	76

viGetNumberOfUnits

```
tmLibdevErr_t viGetNumberOfUnits (
   UInt32   *pNumberOfUnits
);
```

Parameters

`pNumberOfUnits`	Pointer to variable in which to return the number of video-in units available.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function determines the number of video-in peripherals available.

viGetCapabilities

```
tmLibdevErr_t viGetCapabilities (
   pviCapabilities_t   *cap
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.

Description

Provided so that a system resource controller can determine information about the default video-in device before installing it. The **cap** pointer is valid until the video-in library is unloaded. This function will return the capabilities of the default video-in peripheral (i.e. **unit0**).

viGetCapabilitiesM

```
tmLibdevErr_t viGetCapabilitiesM (
   pviCapabilities_t   *cap,
   unitSelect_t         unitName
);
```

Parameters

`cap`	Pointer to a variable in which to return a pointer to capabilities data.
`unitName`	Name of the hardware unit whose capabilities are required.

Return Codes

`TMLIBDEV_OK`	Success.

Description

Provided so that a system resource controller can determine information about the specified video-in unit before installing it. The **cap** pointer is valid until the video-in library is unloaded.

viInstanceSetup

```
tmLibdevErr_t viInstanceSetup(
   Int                 instance,
   viInstanceSetup_t  *setup
);
```

Parameters

`instance`	VI instance to set up.
`setup`	Pointer to a structure containing new parameters.

Return Codes

`TMLIBDEV_OK`	Success.
`VI_ERR_INVALID_DECODER_INT`	Returned if the board has no initialization routine to initialize a VI decoder.
`BOARD_ERR_UNSUPPORTED_STANDARD`	Returned if either video standard or adapter type are not supported by the board video decoder, and any error returned by calls to the board or interrupt library components.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if instance does not match the owner.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assert is triggered if the setup pointer is Null.

Description

This function initializes instance setup parameters, sets VI endianness to the endianness of current execution, programs the VI clock to external mode, initializes the decoder on the board (see Chapter 19, *TMBoard API* in Book 5, *System Utilities*, Part C) by calling the appropriate functions in the board library, and prepares the device for either data streaming mode or video mode.

This function should be called before **viYUVSetup**, or **viRawSetup**.

Related Functions

viOpen, **viOpenM**, **viRawSetup**, and **viYUVSetup**.

viYUVSetup

```
tmLibdevErr_t viYUVSetup(
   Int           instance,
   viYUVSetup_t  *setup
);
```

Parameters

`instance`	VI instance to set up.
`setup`	Pointer to a structure containing new parameters.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if instance does not match the owner.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assert is triggered if the setup pointer is Null.

Description

Sets or changes instance setup parameters in the YUV operation mode. This function checks that every parameter in the **viYUVSetup_t** structure is correct according to the alignment requirements (see **viYUVSetup_t** and the appropriate TriMedia data book), and calls the macro **viSetWIDTH** macro according to the desired mode (full or half resolution).

This function assumes that **viInstanceSetup** has already been called.

Related Functions

viInstanceSetup, viStart.

viRawSetup

```
tmLibdevErr_t viRawSetup(
   Int            instance,
   viRawSetup_t  *setup
);
```

Parameters

`instance`	VI instance to set up.
`setup`	Pointer to a structure containing new parameters.

Return Codes

`TMLIBDEV_OK`	Success. In the debug version there are appropriate alignment and size assertions.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if the instance does not match the owner.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assert is triggered if the setup pointer is Null.

Description

This function sets and changes instance setup parameters in the Raw operation mode.

It assumes **viInstanceSetup** has already been called.

Related Functions

viInstanceSetup, **viStart.**

viOpen

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

Parameters

`instance`	Pointer to a unique instance ID that is set when the default VI device is opened successfully.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NO_MORE_INSTANCES`	Returned when all possible instances have been allocated. Returned by the interrupts library component when the video-in interrupt vector is already in use.

Description

This function attempts to open the default video-in device (i.e. **unit0**), and if successful assigns a unique video-in instance for the caller. This API function should be called first to obtain an instance before any further initialization is performed. It resets the default video-in device using the **viAckRESET** macro, and initializes the associated interrupt.

Related Functions

viOpenM, **viInstanceSetup**, **viClose.**

viOpenM

```
tmLibdevErr_t viOpenM (
   Int          *instance
   unitSelect_t  unitName
);
```

Parameters

`instance`	Pointer to a unique instance ID that is set when the default VI device is opened successfully.
`unitName`	The hardware unit to open.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NO_MORE_INSTANCES`	Returned when all possible instances have been allocated. Returned by the interrupts library component when the video-in interrupt vector is already in use.

Description

This function attempts to open the specified video-in device, and if successful assigns a unique video-in instance for the caller. This API function should be called first to obtain an instance. It resets the required video-in device (**viAckRESET** macro), and initializes the associated interrupt.

Related Functions

viOpen, **viInstanceSetup, viClose.**

viClose

```
tmLibdevErr_t viClose(
   Int   instance
);
```

Parameters

`instance`	VI instance to release.

Return Codes

`TMLIBDEV_OK`	Success.
(other error codes)	Various other errors returned by the interrupt library component when deallocation of the interrupt failed, or when the video encoder's termination function has failed.

Description

This function is used to close an instance. It calls the board's video decoder termination function if one installed. It resets the associated video-in device (**viAckRESET** macro), and closes the interrupt opened by **viOpen/viOpenM.**

Related Functions

viStart, and the board video-in decoder's **term_func**.

viStart

```
tmLibdevErr_t viStart (
   Int   instance
);
```

Parameters

`instance`	VI instance to start.

Return Codes

`TMLIBDEV_OK`	Success.
`VI_ERR_INITIALIZATION_NOT_COMPLETE`	Returned when the initialization is not complete: either **viInstanceSetup** is not called, or no call is made to either **viYUVSetup** or **viRawSetup**.

Description

This function starts the video-in unit associated with the instance. It checks the validity of the instance, and calls the macro **viEnableENABLE**.

Related Functions

viInstanceSetup, **viRawSetup**, **viYUVSetup**, and **viStop**.

viStop

```
tmLibdevErr_t viStop(
   Int   instance
);
```

Parameters

`instance`	VI instance to stop.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This will stop the video-in peripheral associated with the instance. After being called, the associated video interrupt will be disabled.

Related Functions

viStart.

viYUVChangeBuffer

```
void viYUVChangeBuffer(
   Int        instance,
   Pointer    Y,
   Pointer    U,
   Pointer    V
);
```

Parameters

`instance`	VI instance to change.
`Y`	New Y (luminance) buffer pointer.
`U`	New U (chrominanc)e buffer pointer.
`V`	New V (chrominance) buffer pointer.

Return Codes

There is no return code because this function is implemented as a macro.

Description

This function specifies new capture buffers and modifies the pointers directly without instance checking. It uses the macros **viSetY_BASE_ADR**, **viSetU_BASE_ADR**, **viSetV_BASE_ADR** (look at tmVI.h).

viRawChangeBuffer1

```
void viRawChangeBuffer1(
   Int      instance,
   Pointer  buffer
);
```

Parameters

`instance`	VI instance to change.
`buffer`	New buffer pointer for buffer 1.

Return Codes

There is no return code because this function is implemented as a macro.

Description

Sets a new buffer and modifies the pointers directly without instance checking. It uses the macro **viSetBASE1** (refer to tmVI.h).

viRawChangeBuffer2

```
void viRawChangeBuffer2(
   Int       instance,
   Pointer   buffer
);
```

Parameters

`instance`	VI instance to change.
`buffer`	New buffer pointer for buffer 2.

Return Codes

There is no return code since this is implemented as a macro.

Description

Sets a new buffer and modifies the pointers directly without instance checking. It uses the macro **viSetBASE2** (refer to tmVI.h).

viConfigureDecoder

```
tmLibdevErr_t viConfigureDecoder(
   Int      instance,
   UInt32   subaddr,
   UInt32   value,
);
```

Parameters

`instance`	VI instance to configure.
`subaddr`	IIC subaddress to be modified.
`value`	New value to be stored on the sub address.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the configure function for the video-in decoder.
Other errors	Any error returned by the board support package (BSP).

Description

Configures the board's decoder according to the data passed in. This function calls the appropriate function in the board library, depending upon the installed board (currently IREF or DEBUG). For IREF boards, it calls **saa7111Configure**.

Related Functions

The board video decoder's configure function.

viGetColorStandard

```
tmLibdevErr_t viGetColorStandard(
   Int                       instance,
   tmVideoAnalogStandard_t  *colorStandard
);
```

Parameters

`instance`	VI instance.
`colorStandard`	Pointer to the color standard detected. When the function is not implemented, this will be **vasNone**.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	The board does not implement the **getColorStandard** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Detects the board's decoder color standard when this functionality is implemented by the board. (Refer to Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C.)

Related Functions

The board video-decoder's **getColorStandard** function.

viSetBrightness

```
tmLibdevErr_t viSetBrightness(
   Int    instance,
   UInt   level
);
```

Parameters

`instance`	VI instance.
`level`	Brightness level.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setBrightness** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Calls the boards video-decoder **setBrightness** function when this functionality is implemented by the board. (Refer to Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C.)

viSetContrast

```
tmLibdevErr_t viSetContrast(
   Int    instance,
   UInt   level
);
```

Parameters

`instance`	VI instance.
`level`	Contrast level.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	The board does not implement the **setContrast** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Sets the board's decoder contrast level when this functionality is implemented by the board. (Refer to Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C.)

Related Functions

The board video-decoder's **setContrast** function.

viSetHue

```
tmLibdevErr_t viSetHue(
   Int    instance,
   UInt   level
);
```

Parameters

`instance`	VI instance.
`level`	Hue level

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	The board does not implement the **setHue** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Sets the board's decoder hue level when this functionality is implemented by the board. (Refer to Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C.)

Related Functions

The board video-decoder's **setHue** function.

viSetSaturation

```
tmLibdevErr_t viSetSaturation(
   Int     instance,
   UInt    level
);
```

Parameters

`instance`	VI instance.
`level`	Saturation level.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setSaturation** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Sets the board's decoder saturation level when this functionality is implemented by the board. (Refer to Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C.)

Related Functions

The board video-decoder's **setSaturation** function.

viGetVideoStandard

```
tmLibdevErr_t viGetVideoStandard(
   Int                        instance,
   tmVideoAnalogStandard_t   *colorStandard
);
```

Parameters

`instance`	Instance.
`colorStandard`	The color standard detected. When the function is not implemented, this will be **vasNone**.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	The board does not implement the **getVideo-Standard** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the color standard of the decoder.

viGetVSyncFallingEdge

```
tmLibdevErr_t viGetVSyncFallingEdge(
   Int    instance,
   UInt  *lineNumber
);
```

Parameters

`instance`	Instance.
`lineNumber`	Receives the line number.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getVSyncFallingEdge** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the line number in which the falling edge of VSync occurs.

viGetSlicedData

```
tmLibdevErr_t viGetSlicedData(
   Int instance,
   UInt8                *Y,
   UInt8                *U,
   UInt8                *V,
   tmVideoDataService_t  service,
   UInt                  size,
   UInt8                *data,
   UInt8                *dataSize
);
```

Parameters

`instance`	Instance.
`Y, U, V`	Pointers to planar YUV data.
`service`	Teletext data service to extract.
`sizeY`	Size of Y data buffer to search for sliced data.
`data`	Buffer in which to write extracted data.
`dataSize`	Receives the number of extracted bytes.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getSlicedData** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Extracts sliced VBI data. If the decoder inserts sliced data into the video data stream, it will be captured in all three video planes. The pointers to YUV specify the start positions where to extract the data slices.

viGetStatus

```
tmLibdevErr_t viGetStatus(
   Int                    instance,
   tmVideoStatusType_t    type,
   UInt                  *state
);
```

Parameters

`instance`	Instance.
`type`	Describes condition to check.
`state`	Receives current state of specified condition.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getStatus** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get information about the decoder's status (lock, field ID, etc.)

viGetSupportedDataServices

```
tmLibdevErr_t viGetSupportedDataServices(
   Int                    instance,
   tmVideoDataService_t   fieldOne[],
   tmVideoDataService_t   fieldTwo[],
   UInt8                  tblSize
);
```

Parameters

`instance`	Instance.
`fieldOne`	Array receiving supported services for each line of field 1.
`fieldTwo`	Array receiving supported services for each line of field 2.
`tblSize`	Number of lines in field 1 and field 2.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getSupportedDataServices** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get information about data services supported by the decoder's text slicer.

viSetDataServices

```
tmLibdevErr_t viSetDataServices(
   Int                    instance,
   tmVideoDataService_t  fieldOne[],
   tmVideoDataService_t  fieldTwo[],
   UInt8                  tblSize
);
```

Parameters

`instance`	Instance.
`fieldOne`	Specifies the data service for each line of field 1.
`fieldTwo`	Specifies the data service for each line of field 2.
`tblSize`	Number of lines in fieldOne and fieldTwo

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setDataServices** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set up the decoder's text slicer for specific data service in each VBI line.

viGetSlicerLineFlags

```
tmLibdevErr_t viGetSlicerLineFlags(
   Int    instance,
   Bool   fieldOne[],
   Bool   fieldTwo[],
   UInt8  tblSize
);
```

Parameters

`instance`	Instance.
`fieldOne`	Buffer receiving True for every line a service has been found.
`fieldTwo`	Buffer receiving True for every line a service has been found.
`tblSize`	Number of lines in field 1and field 2.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getSlicerLineFlags** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get information from the decoder if specified data services have been found by the decoder’s text slicer.

viEnableSlicing

```
tmLibdevErr_t viEnableSlicing(
   Int    instance,
   Bool   enable
);
```

Parameters

`instance`	Instance.
`enable`	True: enable slicer. False: disable slicer.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **enableSlicing** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Enable the decoder's text slicer.

Note: If the decoder has no separate enable for the slicer, it gets turned on and off by **viSetDataServices**.

viSetSlicerVideoStandard

```
tmLibdevErr_t viSetSlicerVideoStandard(
   Int                      instance,
   tmVideoAnalogStandard_t  standard
);
```

Parameters

`instance`	Instance.
`standard`	Video standard for the slicer.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setSlicerVideoStandard** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the text slicer's video standard.

Note: In most decoders this standard has to match the standard for the active video.

viGetSlicerVideoStandard

```
tmLibdevErr_t viGetSlicerVideoStandard(
   Int                      instance,
   tmVideoAnalogStandard_t  *standard
);
```

Parameters

`instance`	Instance.
`standard`	Receives the slicer's video standard.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getSlicerVideoStandard** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the text slicer's video standard.

Note: In most decoders this standard will match the standard for the active video.

viToggleFieldID

```
tmLibdevErr_t viToggleFieldID(
   Int   instance,
   Bool  toggle
);
```

Parameters

`instance`	Instance.
`toggle`	True: invert field detection. False: do not.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **toggleFieldID** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Toggle the video decoder’s field ID.

viSetSlicerInput

```
tmLibdevErr_t viSetSlicerInput(
   Int    instance,
   UInt   num
);
```

Parameters

`instance`	Instance.
`num`	Input mode for the decoder's slicer.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setSlicerInput** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the slicer's video input.

Note: If the decoder does not support different inputs for active video and the text slicer this function will return an error if **num** differs from the current input for active video.

viGetSlicerInput

```
tmLibdevErr_t viGetSlicerInput(
   Int    instance,
   UInt  *num
);
```

Parameters

`instance`	Instance.
`num`	Pointer to a variable in which to return the current input mode for the decoder's slicer.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getSlicerInput** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the slicer's video input.

Note: If the decoder does not support different inputs for active video and the text slicer this function will return the current input for active video.

viSetVideoColor

```
tmLibdevErr_t viSetVideoColor(
   Int              instance,
   tmVideoColor_t   color,
   UInt             val
);
```

Parameters

`instance`	Instance.
`color`	Defines which color parameter to change.
`val`	The new value.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setVideoColor** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set brightness, contrast, saturation, or hue for the decoder's video input.

viGetVideoColor

```
tmLibdevErr_t viGetVideoColor(
   Int              instance,
   tmVideoColor_t   color,
   UInt            *val
);
```

Parameters

`instance`	Instance.
`color`	Defines which color parameter to get.
`val`	Receives current value.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getVideoColor** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the current value for brightness, contrast, saturation, or hue of the decoder's video input.

viSetAnalogInput

```
tmLibdevErr_t viSetAnalogInput(
   Int   instance,
   UInt  num
);
```

Parameters

`instance`	Instance.
`num`	Defines the new video input.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setAnalogInput** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the decoder's video input.

viGetAnalogInput

```
tmLibdevErr_t viGetAnalogInput(
   Int    instance,
   UInt  *num
);
```

Parameters

`instance`	Instance.
`num`	Receives the current video input.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getAnalogInput** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the decoder’s current video input.

viSetStandard

```
tmLibdevErr_t viSetStandard(
   Int                      instance,
   tmVideoAnalogStandard_t  standard
);
```

Parameters

`instance`	Instance.
`standard`	New video input standard.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setStandard** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the decoder's video input standard.

viSetSourceType

```
tmLibdevErr_t viSetSourceType(
   Int                 instance,
   tmVideoSourceType_t  type
);
```

Parameters

`instance`	Instance.
`type`	New video source type.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setSourceType** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the decoder's video decoder's source type (VCR, TV, ...). Depending on the source type the decoder's SYNC detection will vary.

viGetSourceType

```
tmLibdevErr_t viGetSourceType(
   Int                  instance,
   tmVideoSourceType_t  *type
);
```

Parameters

`instance`	Instance.
`type`	Receives current video source type.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getSourceType** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the decoder's video source type (VCR, TV, ...).

viSetOutputFormat

```
tmLibdevErr_t viSetOutputFormat(
   Int                    instance,
   tmVideoRGBYUVFormat_t  format
);
```

Parameters

`instance`	Instance.
`format`	New video output format.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setOutputFornat** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the decoder's video output format.

viGetOutputFormat

```
tmLibdevErr_t viGetOutputFormat(
   Int                    instance,
   tmVideoRGBYUVFormat_t  *format
);
```

Parameters

`instance`	Instance.
`format`	Receives current video output format.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getOutputFormat** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the decoder’s video output format.

viSetAcquisitionWnd

```
tmLibdevErr_t viSetAcquisitionWnd(
   Int    instance,
   UInt   beginX,
   UInt   beginY,
   UInt   endX,
   UInt   endY
);
```

Parameters

`instance`	Instance.
`beginX`, `beginY`	Horizontal and vertical start of window.
`endX`, `endY`	Horizontal and vertical end of window.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setAcquisitionWindow** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the decoder's video acquisition window. Changing the acquisition window will only affect the video input if the decoder's internal scaler is used.

viGetAcquisitionWnd

```
tmLibdevErr_t viGetAcquisitionWnd(
   Int    instance,
   UInt   *beginX,
   UInt   *beginY,
   UInt   *endX,
   UInt   *endY
);
```

Parameters

`instance`	Instance.
`beginX`, `beginY`	Pointers to variables to get start of window.
`endX`, `endY`	Pointers to variables to get end of window.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getAcquisitionWnd** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the decoder’s video acqusition window.

viGetDefaultAcquisitionWnd

```
tmLibdevErr_t viGetDefaultAcquisitionWnd(
   Int      instance,
   UInt    *beginX,
   UInt    *beginY,
   UInt    *endX,
   UInt    *endY
);
```

Parameters

`instance`	Instance.
`beginX`, `beginY`	Pointers to variables to receive default horizontal and vertical start of window.
`endX`, `endY`	Pointers to variables to get default horizontal and vertical end of window.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getDefaultAcquisitionWnd** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the decoder's default video acquisition window of the active video. The rectangle returned depends on the current video standard and video decoder.

viSetOutputSize

```
tmLibdevErr_t viSetOutputSize(
   Int    instance,
   UInt   width,
   UInt   height
);
```

Parameters

instance	Instance.
width	Specifies new width of the video output window.
height	Specifies new height of the video output window.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setOutputSize** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the decoder's output window. This function can only be used if the video decoder supports an internal scaler.

viSetInterlaceMode

```
tmLibdevErr_t viSetInterlaceMode(
   Int   instance,
   Bool  interlace
);
```

Parameters

`instance`	Instance.
`interlace`	True: use decoder's scaler in interlace mode. False: use the decoder's scaler in field mode.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setInterlaceMode** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the interlace mode of the decoder's internal scaler.

viDisableDecoder

```
tmLibdevErr_t viDisableDecoder(
   Int   instance,
   Bool  disable
);
```

Description

Disable/tristate the decoder's output pins.

Parameters

`instance`	Instance.
`disable`	True: disable decoder. False: enable decoder.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **disableDecoder** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Disable/tristate the decoder's output pins, if supported by the decoder. By default, the decoder is enabled.

viEnablePowerSaveMode

```
tmLibdevErr_t viEnablePowerSaveMode(
   Int   instance,
   Bool  enable
);
```

Parameters

`instance`	Instance.
`enable`	True: turn on power-save mode False: turn off power-save mode.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **enablePowerSaveMode** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Turns the decoder’s power save mode on or off, if the functionality is supported.

viGetGPIOCount

```
tmLibdevErr_t viGetGPIOCount(
   Int    instance,
   UInt  *num
);
```

Parameters

`instance`	Instance.
`num`	Pointer to variable receiving the number of decoder GPIOs.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getGPIOCount** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the number of GPIO pins on the video decoder.

viSetGPIOState

```
tmLibdevErr_t viSetGPIOState(
   Int   instance,
   UInt  pin,
   Bool  state
);
```

Parameters

`instance`	Instance.
`pin`	Number of the decoder's GPIO pin to set.
`state`	True: output high. False: low.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setGPIOState** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the state of a video decoder's GPIO pin.

viGetGPIOState

```
tmLibdevErr_t viGetGPIOState(
   Int    instance,
   UInt   pin,
   Bool  *state
);
```

Parameters

`instance`	Instance.
`pin`	Number of the GPIO pin to get.
`state`	Pointer to variable to get current state. (True = high. False = low.)

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **getGPIOState** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Get the state of a video decoder's GPIO pin.

viOpenVBI

```
tmLibdevErr_t viOpenVBI(
   Int   instance,
   UInt  sampleFreq,
   UInt  startLine,
   UInt  numLines
);
```

Parameters

`instance`	Instance.
`sampleFreq`	VBI sample frequency in Hz.
`startLine`	First VBI line to handle.
`numLines`	Number of VBI lines.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **openVBI** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Open the decoder for raw VBI handling in software. This function sets up the decoder to bypass and/or oversample the appropriate VBI lines. Typical sample frequencies are 13.5 and 27 MHz.

Note: Usage of VBI oversampling will require the usage of hardware syncs with some video decoders.

viEnableVBI

```
tmLibdevErr_t viEnableVBI(
   Int   instance,
   Bool  enable
);
```

Parameters

instance	Instance.
enable	True: enable raw VBI handling False: disable raw VBI handling.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **enableVBI** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Enable the decoder for raw VBI handling. Raw VBI data has to be decoded in software without using a video decoder's text slicer.

viSetVBIMode

```
tmLibdevErr_t viSetVBIMode(
   Int               instance,
   tmVideoVBIMode_t  mode
);
```

Parameters

instance	Instance.
mode	Set mode to raw Y or baseband YUV.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setVBIMode** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the decoder's mode for raw VBI handling. Raw VBI data has to be decoded in software without using a video decoder's text slicer.

viSetSlicerMode

```
tmLibdevErr_t viSetSlicerMode(
   Int                 instance,
   tmVideoSlicerMode_t  mode
);
```

Parameters

`instance`	Instance.
`mode`	Set mode to SAV/EAV or ANC header.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **setSlicerMode** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Set the decoder's mode for sliced VBI data handling. By default SAV/EAV mode will be used. The ancillary data header (ANC) mode requires the video decoder and the video input block to support this feature.

viCloseVBI

```
tmLibdevErr_t viCloseVBI(
   Int   instance
);
```

Parameters

instance	Instance.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the **closeVBI** function for the video-in decoder.
Other errors	Any error returned by the BSP.

Description

Close the decoder's raw VBI data handling.

Video-Out API Data Structures

This section presents the video-out data structures. These data structures are defined in the tmVo.h header file, which also contains the video-out API interface.

Name	Page
voYUVModes_t	78
voRawModes_t	78
voCapabilities_t	79
voRawSetup_t	79
voOverlaySetup_t	80
voYUVSetup_t	81
voInstanceSetup_t	82
voenhChromaKeyingSetup_t	83
voenhClipSetup_t	84
voenhGenLockSetup_t	84

voYUVModes_t

```
typedef enum {
   vo422_COSITED_UNSCALED      = Ø,
   vo422_INTERSPERSED_UNSCALED = 1,
   vo42Ø_UNSCALED              = 2,
   vo422_COSITED_SCALED        = 4,
   vo422_INTERSPERSED_SCALED   = 5,
   vo42Ø_SCALED                = 6
} voYUVModes_t;
```

Description

Enumerate the various YUV image transfer modes for different input data formats and with or without horizontal 2X upscaling.

Note
A full explanation of each of these modes is beyond the scope of this reference. See Section 7 of the appropriate TriMedia data book for more information.

voRawModes_t

```
typedef enum {
   voSTREAM8 = 8,
   voMESSAGE = 9
} voRawModes_t;
```

Description

Enumerates the data streaming modes.

voCapabilities_t

```
typedef struct voCapabilities_t {
   tmVersion_t   version;
   Int           numSupportedInstances;
   Int           numCurrentInstances;
   char          codecName[16];
   UInt32        videoStandards;
   UInt32        adapterTypes;
} voCapabilities_t, *pvoCapabilities_t;
```

Description

Used by the **voGetCapabilities** and **voGetCapabilitiesM** functions, this function provides information about video-out capabilities. This includes the version of the device library, the name of the video codec, and video standards and adapter types supported.

voRawSetup_t

```
typedef struct voRawSetup_t{
   Bool           buf1emptyEnable;
   Bool           buf2emptyEnable;
   voRawModes_t   mode;
   UInt           size1, size2;
   Pointer        base1, base2;
} voRawSetup_t, *pvoRawSetup_t;
```

Fields

`buf1emptyEnable`	Enable interrupt when buffer 1 is empty.
`buf2emptyEnable`	Enable interrupt when buffer 2 is empty.
`size1`, `size2`	Sizes of the two buffers.
`base1, base2`	Pointers to the two buffers.

Description

Describes the settings for the data streaming and message passing modes of the VO peripheral. See the appropriate TriMedia data book for more information.

voOverlaySetup_t

```
typedef struct voOverlaySetup_t{
   Bool      overlayEnable;
   UInt      overlayStartX, overlayStartY;
   UInt      overlayWidth,  overlayHeight;
   UInt      overlayStride;
   Pointer   overlayBase;
   UInt      alpha0, alpha1;
} voOverlaySetup_t, *pvoOverlaySetup_t;
```

Fields

`overlayEnable`	Enable the overlay.
`overlayStartX`, `overlayStartY`	Left upper corner (pixel and line) of the overlay.
`overlayWidth`, `overlayHeight`	Width and height of the overlay.
`overlayStride`	The stride of the overlay (overlay offset in the appropriate TriMedia data book).
`overlayBase`	The base address of the overlay
`alpha0`, `alpha1`	alpha0 is the alpha blend value used in the YUV+alpha mode, when the alpha bit is set to 0; alpha1 is used when the alpha bit (low bit of Y data) is set to 1.

Description

Passed to the **voOverlaySetup** function to initialize the video-out peripheral image overlay for the image passing modes.

voYUVSetup_t

```
typedef struct voYUVSetup_t{
   Bool           buf1emptyEnable;
   Bool           yThresholdEnable;
   voYUVModes_t   mode;
   UInt           imageVertOffset, imageHorzOffset;
   UInt           imageWidth, displayHeight;
   UInt           yThreshold;
   UInt           yStride, uStride, vStride;
   Pointer        yBase, Base, vBase;
} voYUVSetup_t, *pvoYUVSetup_t;
```

Fields

`buf1emptyEnable`	Enable interrupt when buffer 1 is empty.
`yThresholdEnable`	Enable interrupt threshold interrupt.
`mode`	Select the image mode of operation. See the appropriate TriMedia data book for more information.
`imageVertOffset`, `imageHorzOffset`	Vertical and horizontal start of the upper left corner of the output.
`imageWidth`, `displayHeight`	Image width and height in samples.
`yThreshold`	When the **yThresholdEnable** flag is true, an interrupt will be generated when the line counter reaches this value.
`yStride`, `uStride`, `vStride`	Number of bytes from the start of one line to the start of the next line
`yBase`, `uBase`, `vBase`	Pointers to the start of the YUV data.

Description

Passed to the voYUVSetup function to setup the video-out peripheral in image passing mode.

voInstanceSetup_t

```
typedef struct voInstanceSetup_t{
   Bool                        hbeEnable;
   Bool                        underrunEnable;
   UInt32                      ddsFrequency;
   intPriority_t               interruptPriority;
   void                       (*isr)(void);
   tmVideoAnalogStandard_t     videoStandard;
   tmVideoAnalogAdapter_t      adapterType;
} voInstanceSetup_t, *pvoInstanceSetup_t;
```

Fields

`hbeEnable`	Enable interrupts for highway bandwidth errors.
`underrunEnable`	Enable interrupts when an under-run occurs.
`ddsFrequency`	Frequency in Hertz.
`interruptPriority`	VO interrupt priority.
`isr`	Pointer to the interrupt service routine
`videoStandard`	Video standard which the video-out decoder on the board needs to be programmed to.
`adapterType`	The adaptor type (either **vaaCVBS** and **vaaSvideo**) passed to the board's video-out decoder.

Description

This structure is used as the common initializing structure for all video-out modes of operation, including YUV images and raw data streaming modes. It is passed to the **voInstanceSetup** function to perform an initial setup of the video-out peripheral.

voenhChromaKeyingSetup_t

```
typedef struct voenhChromaKeyingSetup_t{
   Bool    keyEnable;
   UInt8   keyY, keyU, keyV;
   UInt8   maskY, maskUV;
} voenhChromaKeyingSetup_t, *pvoenhChromaKeyingSetup_t;
```

Fields

`keyEnable`	Enables the chroma keying feature.
`keyY`, `keyU`, `keyV`	8 bits for each key.
`maskY`, `maskUV`	4 bits for each mask.

Description

When the keyEnable is set, the chroma keying feature is enabled. The overlay values (Y, U and V) are compared to values stored in **keyY**, **keyU**, and **keyV**. Bits that correspond to bits set in **maskY** and **maskUV** are ignored for this comparison. When there is an exact match between the pixel value and the values in **keyY**, **keyU**, and **keyV** (less the bits selected by **maskY** and **maskUV**), then the overlay value is not present in the output stream (full transparency).

voenhClipSetup_t

```
typedef struct voenhClipSetup_t{
   Bool    clipEnable;
   UInt8   highClipUV, lowClipUV, highClipY, lowClipY;
} voenhClipSetup_t, *pvoenhClipSetup_t;
```

Fields

`clipEnable`	Enable the clipping feature.
`highClipUV`, `lowClipUV`	High and low clipping values for U and V.
`highClipY`, `lowClipY`	High and low clipping values for Y.

Description

When **clipEnable** is true, the Y output values are clipped between **lowClipY** and **highClipY**, and the U, V output values are clipped between **lowClipUV** and **highClipUV**.

voenhGenLockSetup_t

```
typedef struct voenhGenLockSetup_t{
   Bool   genLockEnable;
   UInt   slaveDelay;
} voenhGenLockSetup_t, *pvoenhGenLockSetup_t;
```

Fields

`genLockEnable`	When set, enables the TM-1100 genlock feature.
`slaveDelay`	Number of delay cycles in genLock.

Description

TM-1100 genLock works when the video-out is not synchronization master. Frame synchronization is achieved by an external signal on VO_IO2.

Video-Out API Functions

This section describes the TriMedia video-out API functions.

Name	Page
voGetNumberOfUnits	86
voGetCapabilities	87
voGetCapabilitiesM	88
voGetCapabilitiesM	88
voOpen	90
voOpenM	91
voStart	93
voStop	94
voYUVSetup	95
voOverlaySetup	97
voRawSetup	98
voYUVChangeBuffer	99
voOverlayChangeBuffer	100
voRawChangeBuffer1	101
voRawChangeBuffer2	102
voConfigureEncoder	103
voSetBrightness	104
voSetHue	105
voSetSaturation	106
voenhStart	107
voenhClipSetup	107
voenhChromaKeyingSetup	108
voenhGenLockSetup	108

voGetNumberOfUnits

```
tmLibdevErr_t voGetNumberOfUnits (
   UInt32   *pNumberOfUnits
);
```

Parameters

`pNumberOfUnits`	Pointer to a variable in which to return the number of hardware units available.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function determines the number of video-out peripherals which are available.

voGetCapabilities

```
tmLibdevErr_t voGetCapabilities(
   pvoCapabilities_t   *cap
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assert is triggered if **cap** is null.
`TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW`	In the debug version of the library, this assert is triggered if **cap** is null.

Description

Sets a pointer to default video-out (**unit0**) capability structure.

voGetCapabilitiesM

```
tmLibdevErr_t voGetCapabilitiesM (
   pvoCapabilities_t   *cap,
   unitSelect_t         unitName
);
```

Parameters

`cap`	Pointer to a variable in which to return a pointer to capabilities data.
`unitName`	Name of the hardware unit whose capabilities are required.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW`	In the debug version of the library, this assert is triggered if cap is null.

Description

Provided so that a system resource controller can determine information about the specified video-out unit before installing it.

voInstanceSetup

```
tmLibdevErr_t voInstanceSetup(
   Int                  instance,
   voInstanceSetup_t   *setup
);
```

Parameters

`instance`	Current instance of VO.
`setup`	Setup pointer to buffer that holds new parameters.

Return Codes

`TMLIBDEV_OK`	Success.
`VO_ERR_INVALID_ENCODER_INT`	Returned if the board has no initialization routine to initialize a VO decoder.
`BOARD_ERR_UNSUPPORTED_STANDARD`	Returned if either video standard or adapter type are not supported by the board video encoder, and any error returned by calls to the board or interrupt library components.
`BOARD_ERR_UNSUPPORTED_ADAPTER`	Returned if either video standard or adapter type are not supported by the board video encoder, and any error returned by calls to the board or interrupt library components.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if instance does not match the owner.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assert is triggered if **setup** is null.

Description

Sets or changes the setup parameters for a specific instance. It sets VO endianness to that of current execution, initializes the encoder on the board according to the board type (see Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C) and prepares for either data or video streaming mode.

Call this function before calling **voYUVSetup** or **voRawSetup**.

Related Functions

voOpen, **voOpenM**, **voYUVSetup**, **and voRawSetup.**

voOpen

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

Parameters

`instance`	Pointer to a unique instance ID which is set when the default VO device is opened successfully.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NO_MORE_INSTANCES`	Returned when all possible instances have been given out. Returned by the interrupts library component when the video-out interrupt vector is already in use.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assert is triggered if instance is null.

Description

Assigns a unique video-out instance for the caller; the default video-out peripheral is used (**unit0**). It opens and sets up an interrupt for the video-out unit with the **intOpen** and **intInstanceSetup**.

Related Functions

voOpenM, **voInstanceSetup**, **voClose.**

voOpenM

```
tmLibdevErr_t voOpenM (
   Int           *instance
   unitSelect_t   unitName
);
```

Parameters

`instance`	Pointer to a unique instance ID which is set when the desired VO device is opened successfully.
`unitName`	The hardware unit to open.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NO_MORE_INSTANCES`	Returned when all possible instances have been given out. Returned by the interrupts library component when the video-out interrupt vector was already in use.

Description

This function attempts to open the specified video-out device, and if successful assigns a unique instance for the caller. This API should be called first to obtain an instance. It resets the required video out device (**voAckRESET** macro), and initializes the interrupt for the device.

Related Functions

voOpen, **voInstanceSetup**, **voClose**.

voClose

```
tmLibdevErr_t voClose(
   Int   instance
);
```

Parameters

`instance`	Video-out instance to be released.

Return Codes

`TMLIBDEV_OK`	Success. Various other errors returned by the interrupt library component when deallocation of the interrupt failed, or when the video decoder's termination function has failed.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if instance does not match the owner.

Description

Deallocates the video-out instance, and uninstalls its handler when it has one, closes the interrupt opened by **voOpen/voOpenM** with **intClose**, resets the video-out unit, calls the board's video encoder termination function when that function is installed.

Related Functions

voStart, and the board video-out encoder’s termination function.

voStart

```
tmLibdevErr_t voStart(
   Int   instance
);
```

Parameters

`instance`	Video-out instance to start.

Return Codes

`TMLIBDEV_OK`	Success.
`VO_ERR_INITIALIZATION_NOT_COMPLETE`	Either **voInstanceSetup** is not called, or no call is made to either **voYUVSetup** or **voRawSetup**.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if instance does not match the owner.

Description

Starts the video-out unit. After checking that video-out is properly initialized, it calls the macro **voEnableENABLE**.

Related Functions

voInstanceSetup, **voRawSetup**, **voYUVSetup**, and **voOverlaySetup**.

voStop

```
tmLibdevErr_t voStop(
   Int   instance
);
```

Parameters

`instance`	Video-out instance to stop.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if instance does not match the owner.

Description

Stop the video-out peripheral. Interrupts will no longer be generated, do not release the instance and leave the setup as is. This function calls the macro **voDisableENABLE**.

Related Functions

vostart.

voYUVSetup

```
tmLibdevErr_t voYUVSetup(
   Int            instance,
   voYUVSetup_t   *setup
);
```

Parameters

`instance`	Video-out instance.
`setup`	Setup pointer to a structure holding the setup parameters. Refer to **voYUVSetup_t**.

Return Codes

`TMLIBDEV_OK`	Success. In the debuggable version of the libdev library there are appropriate alignment and size assertions given.
`VO_ERR_MODE_SIZE`	In the debug version, this assertion is triggered if **setup->mode** is a data streaming mode (**voRawMode_t**), and not an image transfer mode (**voYUVMode_t**).
`VO_ERR_IMAGE_HOFF_SIZE`	In the debug version, this assertion is triggered if **(setup->imageWidth + setup->imageHorzOffset)** is bigger than the standard frame width, set up in **VOinstanceSetup**. This assertion is also triggered if the value of **setup->imageHorzOffset** is represented on more than 8 bits.
`VO_ERR_IMAGE_VOFF_SIZE`	Save definition as **VO_ERR_IMAGE_MOFF_SIZE**, but in the vertical direction.
`VO_ERR_IMAGE_WIDTH_SIZE`, `VO_ERR_IMAGE_HEIGHT_SIZE`	These are assertions, triggered if the values of **setup->imageWidth** or **setup->imageHeight** is represented on more than 12 bits.
`VO_ERR_Y_THRESHOLD_SIZE`	This assertion is triggered if the value of **setup->yThreshold** is represented on more than 12 bits.
`VO_ERR_Y_OFFSET_SIZE`	This assertion is triggered if the value of **setup->yStride** is represented on more than 16 bits.
`VO_ERR_U_OFFSET_SIZE`	This assertion is triggered if the value of **setup->uStride** is represented on more than 16 bits.

`VO_ERR_V_OFFSET_SIZE`	This assertion is triggered if the value of **setup->vStride** is represented on more than 16 bits.
`VO_ERR_Y_BASE_ADR_SIZE`	This assertion is triggered if the value of **setup->yBase** is represented on more than 32 bits.
`VO_ERR_U_BASE_ADR_SIZE`	This assertion is triggered if the value of **setup->uBase** is represented on more than 32 bits.
`VVO_ERR_V_BASE_ADR_SIZE`	This assertion is triggered if the value of **setup->vBase** is represented on more than 32 bits.

Description

This function prepares video-out according to the parameters given in ***setup**; this function should be used when image transfer mode is required. Refer to **voYUVSetup_t** for further information. This function assumes that a call to **voInstanceSetup** has been made.

Related Functions

voInstanceSetup, **voStart**.

voOverlaySetup

```
tmLibdevErr_t voOverlaySetup(
   Int                  instance,
   voOverlaySetup_t   *setup
);
```

Parameters

`instance`	Video-out instance to setup.
`setup`	Setup pointer to a structure holding the setup parameters. Refer to **voOverlaySetup_t**.

Return Codes

`TMLIBDEV_OK`	Success. In the debuggable version of the libdev library there are appropriate alignment and size assertions given.
`TMLIBDEV_ERR_NULL_PARAMETER`	Triggered if setup is null.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if instance does not match the owner.
`VO_ERR_OL_WIDTH_SIZE`	This assertion is triggered if **(setup->overlayStartX + setup->overlayWidth)** is bigger than the standard frame width. This assertion is also triggered if the value of **setup->overlayWidth** is represented on more than 12 bits.
`VO_ERR_OL_HEIGHT_SIZE`	Same definition as above, but in the vertical direction.
`VO_ERR_ALPHAØ_SIZE`	This assertion is triggered if the value of **setup->alpha0** is represented on more than 8 bits.
`VO_ERR_ALPHA1_SIZE`	This assertion is triggered if the value of **setup->alpha1** is represented on more than 8 bits.
`VO_ERR_OL_BASE_ADR_SIZE`	This assertion is triggered if **setup->overlayBase** is represented on more than 32 bits.
`VO_ERR_OL_OFFSET_SIZE`	This assertion is triggered if **setup->overlayStride** is represented on more than 16 bits.

Description

Sets up overlay-related parameters (see **voOverlaySetup_t** definition for further information).

voRawSetup

```
tmLibdevErr_t voRawSetup(
   Int              instance,
   voRawSetup_t   *setup
);
```

Parameters

`instance`	Video-out instance to setup.
`setup`	Setup pointer to a structure holding the setup parameters. Refer to **voRawSetup_t**.

Return Codes

TMLIBDEV_OK	Success. In the debuggable version of the libdev library, there are appropriate alignment and size assertions given.
VO_ERR_MODE_SIZE	This assertion is triggered if **setup->mode** is an image transfer mode (**voVUVMode_t**), not a data streaming mode (**voRawMode_t**).
`VO_ERR_BASE1_SIZE`	This assertion is triggered if **setup->base1** is not 64-bit aligned addresses, and if the value of **setup->base1** is represented on more than 32 bits.
`VO_ERR_BASE2_SIZE`	This assertion is triggered if **setup->base2** is not 64-bit aligned addresses, and if the value of **setup->base2** is represented on more than 32 bits.
`VO_ERR_SIZE1_SIZE`	This assertion is triggered if **setup->size1** is not 64-bit aligned addresses, and if the value of **setup->size1** is represented on more than 32 bits.
`VO_ERR_SIZE2_SIZE`	This assertion is triggered if **setup->size2** is not 64-bit aligned addresses, and if the value of **setup->size2** is represented on more than 32 bits.

Description

Used to set the data streaming mode parameters. Refer to **voRawSetup_t** definition for further information.

It assumes **voInstanceSetup** has already been called.

Related Functions

voInstanceSetup, and **voStart**.

voYUVChangeBuffer

```
tmLibdevErr_t voYUVChangeBuffer(
   Int        instance,
   Pointer    Y,
   Pointer    U,
   Pointer    V
);
```

Parameters

`instance`	Video-out instance.
`Y`	New luminance buffer pointer.
`U`	New U chrominance buffer pointer.
`V`	New V chrominance buffer pointer.

Return Codes

Because this function is implemented as a macro, it does not return an error code.

Description

This function sets a new display buffer and modifies the MMIO registers directly without instance checking. It calls the three macros: **voSetY_BASE_ADR**, **voSetU_BASE_ADR**, **voSetV_BASE_ADR** (refer to tmVO.h).

voOverlayChangeBuffer

```
tmLibdevErr_t voOverlayChangeBuffer(
   Int        instance,
   Pointer    Y,
   Pointer    U,
   Pointer    V
);
```

Parameters

`instance`	Video-out instance.
`Y`	New luminance buffer.
`U`	New U chrominance buffer.
`V`	New V chrominance buffer.

Return Codes

Because this function is implemented as a macro it does not return an error code.

Description

This function sets a new overlay buffer and modifies the mmio registers directly without instance checking. It calls the macro **voSetOL_BASE_ADR** (refer to tmVO.h).

voRawChangeBuffer1

```
tmLibdevErr_t voRawChangeBuffer1(
   Int       instance,
   Pointer   buffer,
   UInt      size
);
```

Parameters

`instance`	Video-out instance.
`buffer`	New buffer pointer.
`size`	New size.

Return Codes

Because this function is implemented as a macro, it does not return an error code.

Description

This function sets a new data-transfer buffer and modifies the mmio registers directly without instance checking. This calls the macros **voSetBASE1** and **voSetSIZE1** (refer to tmVO.h).

voRawChangeBuffer2

```
tmLibdevErr_t voRawChangeBuffer2(
   Int        instance,
   Pointer    buffer,
   UInt       size
);
```

Parameters

`instance`	Video-out instance.
`buffer`	New buffer pointer.
`size`	New size.

Return Codes

Because this function is implemented as a macro, it does not return an error code.

Description

This function sets a new data transfer buffer and modifies the MMIO registers directly without instance checking. This calls the macros **voSetBASE2** and **voSetSIZE2** (refer to tmVO.h).

voConfigureEncoder

```
tmLibdevErr_t voConfigureEncoder(
   Int       instance,
   UInt32    subaddr,
   UInt32    value
);
```

Parameters

`instance`	Video-out instance.
`subaddr`	IIC subaddress of the encoder.
`value`	New value to be stored on the subaddress.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the configure function for the video-out encoder.
Other errors	Any error returned by the BSP.

Description

Calls the appropriate board's configure function with the parameters above. The function depends on the board (see Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C).

Related Functions

The board video encoder's configure function.

voSetBrightness

```
tmLibdevErr_t voSetBrightness(
   Int     instance
   UInt    brightLevel
);
```

Parameters

`instance`	Video-out instance.
`level`	Brightness level

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the setBrightness function for the video-out encoder.
Other errors	Any error returned by the BSP.

Description

Detects the board video-encoder's **setBrighness** function and sets brightness to **brightLevel** when this functionality is implemented by the board (see Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C).

voSetHue

```
tmLibdevErr_t voSetHue(
   Int    instance,
   UInt   hueLevel
);
```

Parameters

instance	Video-out instance.
level	Hue level.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the setHue function for the video-out encoder.
Other errors	Any error returned by the BSP.

Description

Sets the board's encoder hue level when this functionality is implemented by the board (see Chapter 19, *TMBoard API*, in Book 5, *System Utilities*, Part C).

Related Functions

The board video-encoder's **setHue** function.

voSetSaturation

```
tmLibdevErr_t voSetSaturation(
   Int    instance,
   UInt   saturationLevel
);
```

Parameters

`instance`	Video-out instance.
`saturationLevel`	Saturation level.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	Returned when the board does not implement the setSaturation function for the video-out encoder.
Other errors	Any error returned by the BSP.

Description

Sets the board’s encoder saturation level when this functionality is implemented by the board.

Related Functions

The board video-encoder's **setSaturation** function.

voenhStart

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

Parameters

`instance`	Enhanced video-out instance.

Return Codes

The function can return any error code generated by Enhanced Video Out.

Description

The start and stop function for enhanced video-out unit.

voenhClipSetup

```
tmLibdevErr_t voenhClipSetup
   Int                instance,
   voenhClipSetup_t   *setup
);
```

Parameters

`instance`	Enhanced video-out instance.
`setup`	Pointer to buffer holding new parameters.

Return Codes

The function can return any error code generated by Enhanced Video Out.

Description

Set up the higher and lower clipping values for Y, U, and V.

voenhChromaKeyingSetup

```
tmLibdevErr_t voenhChromaKeyingSetup
   Int                         instance,
   voenhChromaKeyingSetup_t   *setup
);
```

Parameters

`instance`	Enhanced video-instance.
`setup`	Pointer to buffer holding new parameters.

Return Codes

The function can return any error code generated by Enhanced Video Out.

Description

Sets up the chroma key and mask values for the Y, U, and V components.

voenhGenLockSetup

```
tmLibdevErr_t voenhGenLockSetup
   Int                    instance,
   voenhGenLockSetup_t   *setup
);
```

Parameters

`instance`	Enhanced video-out instance.
`setup`	Pointer to buffer holding new parameters.

Return Codes

The function can return any error code generated by Enhanced Video Out.

Description

Sets up the GenLock related parameters.

Chapter 2

Video Capturer (VcapVI) API

VcapVI API Overview

The TriMedia Video Capturer (VcapVI) is an implementation of a TSSA-compliant video input driver. It is similar to the video digitizer (VdigVI), but provides additional VBI information. An OL layer only is provided. VcapVI delivers video data and sliced vertical blanking information (VBI) data to downstream TSSA components. The component provides the functionality to do slicing in a separate task. To reduce the processor load, hardware VBI slicing is used whenever possible. The video capturer decides for every VBI line in the video buffer whether it will do the slicing in software or in hardware, depending on the hardware capabilities of the available video-ADC.

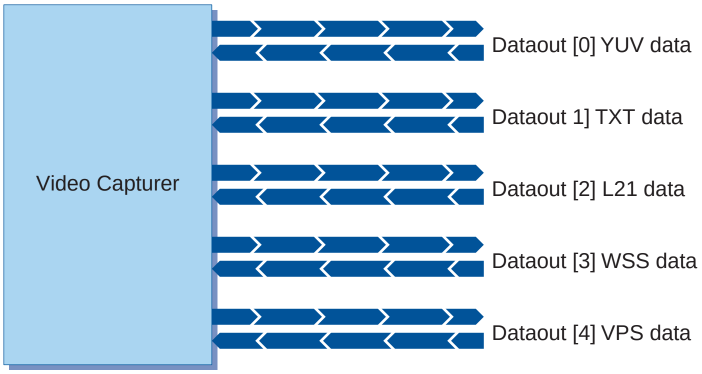


Figure 1 Structure of the Video Capturer

The video capturer is a high-level library using the video-in device library. Using the board ID, the device library can control the external video analog/digital converter chip. Some of these chips also provide hardware-supported slicing. The VcapVI supports software slicing for the following services: European Teletext, NABTS, European Closed Caption, US Closed Caption, Wide Screen Signalling (WSS), and Video Programming System (VPS). All the supported hardware slicing is chip dependent.

The application does not have to worry about the required interrupt service routine in order to handle the hardware video-in events—this is taken care of by the video capturer. See Figure 2, following.

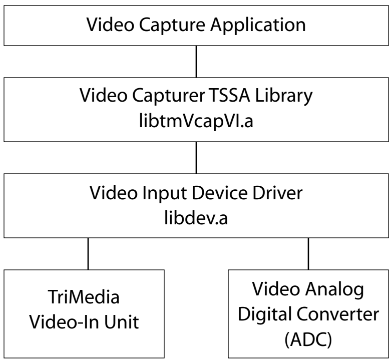

Video Capture Application
Video Capturer TSSA Library
libtmVcapVI.a
Video Input Device Driver
libdev.a
TriMedia
Video-In Unit
Video Analog
Digital Converter
(ADC)

Figure 2 Video Capturer Architecture

Using the VcapVI API

The TriMedia Video Capturer API library name is libtmVcapVI.a. For using the Video Capturer API, the tmolVcapVI.h header file has to be included.

VcapVI supports data streaming operation using the dataoutFunc callback within the video-in interrupt service routine. By default, the instance will use the dataoutFunc function supplied by the Defaults library. A typical flow of control is shown in Figure 3.

First, the application obtains the capabilities of the component and the hardware unit using **tmolVcapVIGetCapabilities** or **tmolVcapVIGetCapabilitiesM**. The "M" function has to be used if the application needs to specify a unit other than the default (default is unit number zero). The acquired information will be used by the format manager to ensure that the two components being connected are compatible.

In order to use the VcapVI functionality, an instance of the video capturer must be created by calling **tmolVcapVIOpen**. If the underlying hardware has more than one video-in unit, the application has to specify which unit the instance has to be connected to. This can be done in a call to **tmolVcapVIOpenM**. For instance, the TriMedia family TM-1 only supports one video-in unit, the TM-2 two. Only one instance can be attached to a specific unit.

The application can then obtain a pointer to the instance setup structure using **tmolVcapVIGetInstanceSetup**; this structure is automatically created when the instance is opened. It contains default values for the opened instance. The application can then setup fields such as the video standard and adapter type, and pass the structure to the instance with **tmolVcapVIInstanceSetup**. The description of tmolVcapVIGetInstanceSetup gives information about the default values.

The **tmolVcapVIStart** function begins the data streaming operation. The capturer will use the **dataoutFunc** callback to obtain an empty packet where the captured video data will be stored. If no empty packet is available the video capturer does not start with the data streaming and the start function returns with an error code. After capturing one field, the capturer will attempt to acquire another empty packet using the **dataoutFunc** callback. If successful, it will send out the packet with the recently acquired image to the connected downstream component. If the acquisition of an empty packet fails, the capturer will simply use the packet which it has in its possession to store the next field. This overrun condition is signalled by the instance using the **progressFunc** callback. How the data slicing works is described later.

Data streaming can be terminated by calling **tmolVcapVIStop** at any time. This will stop the video-in device, and expel the packet currently being held by the instance. The application can release the instance by calling **tmolVcapVIClose**.

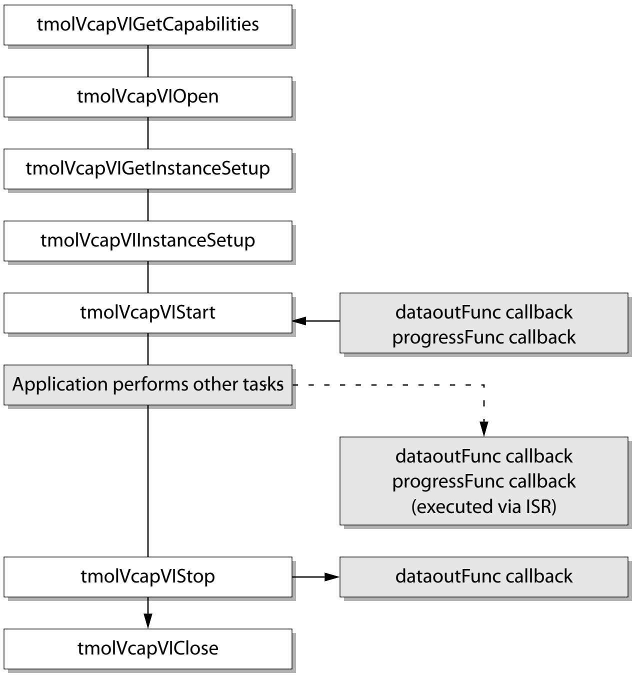


Figure 3 Data streaming flow control

The VcapVI component contains an extra task which takes care about the slicing functionality. This task is triggered by a received full video packet signal from the video-in

ISR. Once a full video packet has arrived, data slicing is applied. After that the video and VBI data packets are sent out.

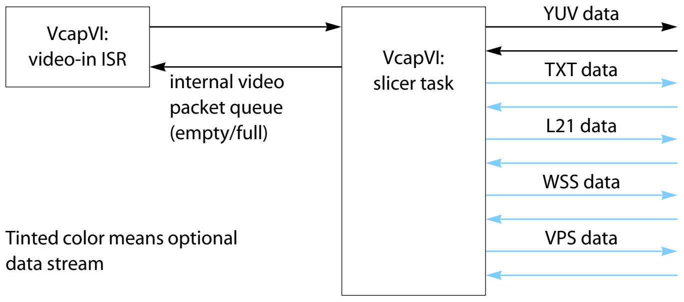


Figure 4 VcapVI internal and external data flow

Integrated Slicing Mechanism

The application specifies what VBI data services have to be sliced by VcapVI, by setting the data service flags (of type **dataService_t)** in the **field1Lines** and **field2Lines** arrays in the **tmolVcapVIInstanceSetup_t** structure. The array index plus one represents the line number where the corresponding data service have to be sliced.

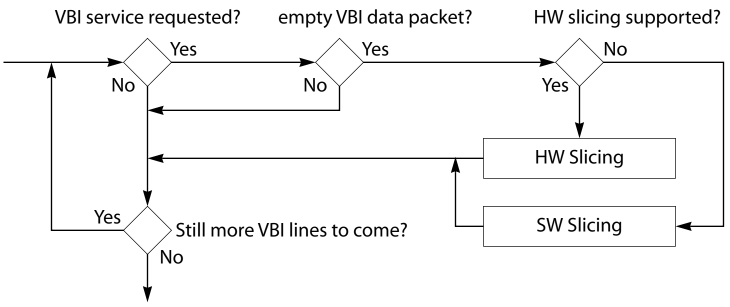


Figure 5 Decision graph concerning the slicing functionality, which is part of the slicer task of the VcapVI component

The device library gives the VcapVI component information of all provided data services which have hardware support. Since the data service flags can be OR'd, multiple VBI data services can be supported per line. Using this capability information the VcapVI component decides what requested service is sliced by software and by hardware.

The mechanism shown in Figure 5 works as follows. First, the software slicer gets a full video packet. Then it looks in the format information of the video packets, to see if it contains VBI information. If yes, the slicing will be started with the first available VBI data line. If no empty VBI data packet has been received before, the corresponding service is being skipped. This condition is reported to the application by calling the progress function. If the service dependent empty VBI data packet is available, software or hardware slicing is being performed.

Software Slicing

Software slicing uses DSPCPU resources. No TriMedia internal hardware block is being used to do this task. Special optimizations have been made to do the software slicing as fast as possible. Table 1 gives an overview about the software slicer performance.

Table 1 Performance overview of how many cycles are required to slice one line VBI data of field which was previously been captured by the VcapVI component.

VBI data service	Number of cycles to slice one line of VBI data
DT_EU_TELETEXT	23K cycles
DT_US_NABTS	23K cycles
DT_EU_CLOSECAPTION	17K cycles
DT_US_CLOSECAPTION	17K cycles
DT_WSS	20K cycles
DT_VPS	21K cycles

These performance numbers only give an amount of cycles required by the software slicing function itself. Additional cycles have to be added which are caused by the operating system (task switch) and the VcapVI API layer itself. Since the video packet can only be sent out after performing the slicing the slicing delay also influences the video delay.

Hardware Slicing

Depending on the information the video-in device library gets from the hardware, hardware data slicing is used. This kind of data slicing is also done in the task associated with VcapVI. Depending on the connected hardware, different kinds of mechanisms are used by the device library to retrieve the sliced data from the captured video data. Some chips (e.g., the SAA7111) deliver the sliced data via the IIC bus. This means that additional interrupt traffic has to be taken into account. The device library description provides information how the sliced data is retrieved from the connected ADC device. Because of that, no performance information can be provided in this documentation, since the VBI data acquisition is all being done in the video device library.

Line counting issues

To handle the vertical blanking interval (VBI) data in a generic way—VBI data is always at the beginning of a video packet—the following line counting issues are addressed. The 60-field and 50-field video systems are using different line counting bases, the 50 Hz system for instance starts counting on the serration pulses and the 60 Hz systems start counting at the first equalization pluses. The equalization and serration pulses can be seen in the vertical timing diagrams Figure 6 and Figure 7. Those pulses have twice the line frequency and are located in the vertical blanking interval.

Table 2 Field interval definitions according to CCIR 656

Field	SAV/EAV bit	50 Hz (625 lines)	60 Hz (525 lines)
V-digital field blanking			
Field 1	Start (V=1)	Line 624	Line 1
	Finish (V=0)	Line 23	Line 10
Field 2	Start (V=1)	Line 311	Line 264
	Finish (V=0)	Line 336	Line 273
F-digital field identification			
Field 1	F = 0	Line 1	Line 4
Field 2	F = 1	Line 313	Line 266

In 50 Hz systems, the startY field have an allowed lowest value of -2, to get the content of the first equalization pulse group, since 50 Hz line counting starts after this first equalization pulse group, and VdigVI always has to deliver the VBI data at the beginning of the video buffer. Since NTSC counting starts at the beginning of the first equalization pulses, the lowest allowed value in this system is zero. This makes the use of the digitizer component as easy as possible. e.g. an application wants to get line 21 data additional to the active video, the **startY** value has to be set to 21 in either frequency (50 Hz or 60 Hz).

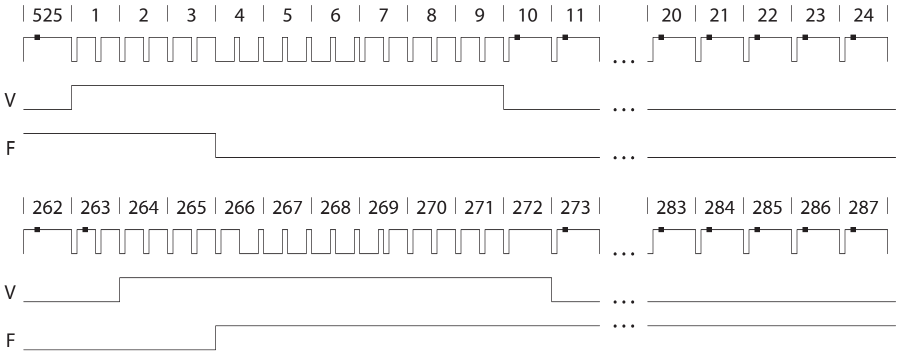


Figure 6 Vertical timing diagram for 60 Hz and its corresponding SAV/SEAV bits. V: vertical sync. F: field flag

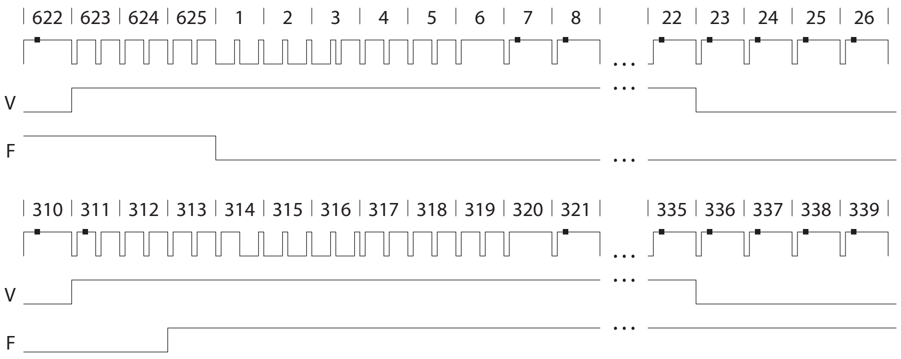


Figure 7 Vertical timing diagram for 50 Hz and its corresponding SAV/EAV bits.
V: vertical sync. F: field flag

The video-in hardware block starts counting lines when the V bit in the SAV/EAV codes goes to zero. The CCIR-656 standard defines the following lines where this happens: 50 Hz, line 23 and 60 Hz, line 10. This leads to the following conclusion: if the digitizer has to capture before these lines (always the case for VBI capturing in 50 Hz systems), a much higher line count has to be put in the hardware registers to start capturing in the VBI interval. The first field has one line more than the second field (313/312 in 625-line systems and 263/262 in 525-line systems). This leads to the restriction that the resulting line (number in the register) cannot be 313 or 263 in 50 Hz or 60 Hz systems respectively, because the internal counter of the VI-block only gets reached in the first field, but never in the second field. This fact causes one little restriction. The internal start value of capturing never can be 313/263. But if an application wants to get the content of this line it has to start with the capturing one line before.

The fields **activeVideoStartX**, **activeVideoStartY**, **activeVideoEndX**, and **activeVideoEndY** have nothing to do with the defined line counting in the 60 Hz and 50 Hz systems. Those values only represents offset values where a downstream component can find the active video area by taking into account how the video data are organized in the video buffer. In interlaced systems the values (times stride) directly lead to the address of the active video data. In field-in-field systems the content of the structure fields have to be divided by two in order to apply the same calculation like in the interlaced case to get to the right active video addresses. That is why these field have a range from 0 to 576 or from 0 to 480 respectively.

Cache Coherency

The application has to use the **tsaIODescSetupFlagInvalidateDataout** flag for creation of the data queue between video capturer and the downstream component Other cache coherency issues are automatically handled by the tsaDefaults library when VcapVI is connected to another TSSA component. When being connected to a non TSSA component, the component needs to invalidate the video data before use by the DSPCPU.

VcapVI Inputs and Outputs

The video capturer is a data source, therefore, it has no input pin and provides besides the video output pin several VBI data pins. The output format of the video pin can be specified using the instance setup function. The field **pOutputFormat** has to be filled by the application. The value must be a pointer to a **tmVideoFormat_t** structure. The video capturer asserts if the installed format does not match with the format installed on the output queue. The video capturer checks if the containing parameters are supported by the library. Currently only the TV standards NTSC and PAL and its related parameters are supported.

The VBI data output pins have specified output formats. All of these pins send packets out containing data in the 'Generic' format. Usually, if the requested data service is available, full VBI data packets are sent out directly after the video packet is sent out.

The video capturer configures the number of output pins automatically, depending on downstream components being connected or not by the application. If no empty VBI data packet is available and the video capturer got a full video packet, no slicing is performed and the progress function will be called to signal the application that the component lost a VBI data packet.

A downstream component connected to the video output pin is a requirement.

If no VBI data is available, no full VBI data packets are sent out until decodeable data is detected in the video signal again.

The following table Table 3 gives an overview of the provided output pins and its required buffers and it's sizes, which have to be provided by the application.

Table 3 Output pins, the delivered data type, and the required buffer sizes

Pin ID	Supports	Buffer Size
MAIN_OUTPUT	video data	buffers[0]: vdfFieldInFrame: height × stride vdfFieldInField: height × stride / 2 buffers[1]: vdfFieldInFrame: height × stride / 2 vdfFieldInField: height × stride / 4 buffers[2]: vdfFieldInFrame: height × stride / 2 vdfFieldInField: height × stride / 4
TXT_OUTPUT	DT_EU_TELETEXT	buffers[0]: 42 bytes
	DT_US_NABTS	buffers[0]: 33 bytes

Table 3 Output pins, the delivered data type, and the required buffer sizes

Pin ID	Supports	Buffer Size
L21_OUTPUT	DT_EU_CLOSECAPTION	buffers[0]: 4 bytes
	DT_US_CLOSECAPTION	buffers[0]: 4 bytes
WSS_OUTPUT	DT_WSS	buffers[0]: 14 bytes
VPS_OUTPUT	DT_VPS	buffers[0]: 26 bytes

Packet Formats

Several packet formats are being used by the video capturer. The format on the main output pin is described next.

Main Output Pin Format

The video capturer uses the standard packet data types **tmAvPacket_t** defined in the tmAvFormats.h include file. The captured YUV data is stored in three buffers, with the Y data contained in **buffer[0]**, and the UV data contained in **buffer[1]** and **buffer[2]** respectively.

Each packet contains a header structure providing information concerning the packet data. The format field is a pointer to a **tmVideoFormat_t** structure which specifies the format and the image size. There are restrictions on the type of video formats that can be used by the video capturer. These will be described next.

The main image output packet can be either **vdfYUV422Planar** or **vdfYUV422Interspersed**. No YUV420 format is supported by the video-in unit. If the video capturer is used also for capturing of vertical blanking interval data, the vdfYUV422Planar flag needs to be used (in this mode the video in hardware does not perform any filtering of the incoming data). The **pOutputFormat** field in the instance setup structure should be initialized with the following values:

Table 4 Main Output Pin Format

Field	Set by	Value
dataClass	App	avdcVideo
dataType	App	vtfYUV
dataSubtype	App	vdfYUV422Planar or vdfYUV422Interspersed
description	App	vdfFieldInFrame or vdfFieldInField
imageWidth	App	Width of video frame in pixels (luminance)
imageHeight	App	Height of video frame in lines (luminance)

Table 4 Main Output Pin Format

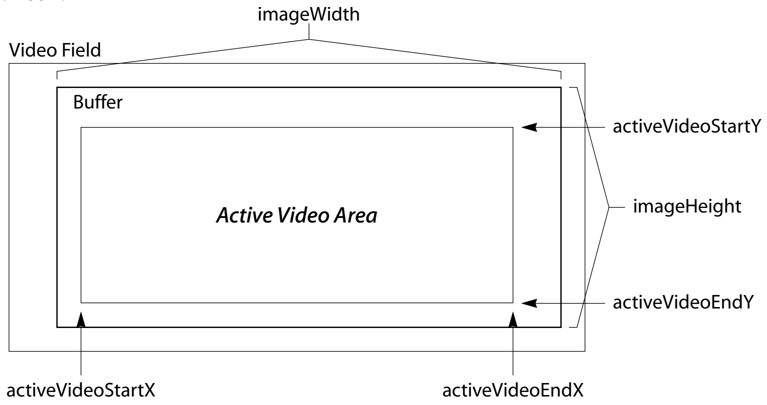
Field	**Set by**	**Value**
imageStride	App	Stride of video frame in bytes (luminance)
activeVideoStartX	VcapVI	Defines pixel offset in horizontal direction from start of video buffer to beginning of active video.
activeVideoStartY	VcapVI	Defines number of lines from start of video buffer to beginning of active video
activeVideoEndX	VcapVI	Defines end of active video area in number of pixels within the video buffer. It is an absolute position.
activeVideoEndY	VcapVI	Defines end of active video area in number of lines within the video buffer. It is an absolute position.
videoStandard	VcapVI	Defines analog video standard that served as source of digitized image

The **description** field of the format structure is set by the video capturer automatically depending on the instance setup field **interlaced**, but it still checks if the resulting description matches with the previously installed format of the output queue.

Table 5 The description field is set by the VcapVI library during the instance setup

Description	**Meaning**
interlaced == True	vdfFieldInFrame Only one field is written in the packet buffer. Two consecutive lines of one field have one not updated line of the other field in between. The packet is sent out field based.
interlaced == False	vdfFieldInField Only one field is written in the packet buffer. No space is between two consecutive lines of one field. The format used in the packet being sent out is field-based.

In field based operation every packet is marked in the flags field of the header structure if the packet contains the second field (**avhField2**) or not (**!avhField2**). Using the description and field information, a downstream component knows which data bytes are valid in the packet and which are not. In case of description **vdfFieldInFrame** it is possible to build up a complete frame by sending the half-filled frame back from the downstream component to the capturer which inserts during capturing the missing field in the lines between.



The fields **activeVideoStartX**, **activeVideoStartY**, **activeVideoEndX**, **activeVideoEndY**, and **videoStandard** are also set by the video capturer automatically. However, in this case it does not check if they match the current queue setup. Depending what analog video standard was chosen the video capturer sets these fields accordingly. Using the location of the active video area, a downstream component has access to additional information, such as VBI inserted data, which is transmitted by the video signal in parallel. Data services currently not being sliced by the VcapVI component can be handled downstream, since all information to do so is available in the packet. There is one restriction to this. Some video-ADC's place hardware sliced data in the video stream and thus override the original VBI information. In this case the original VBI data is lost.

TXT Output Packet Format

Compared to the main output packet format, the VBI data packets can be described using the standard tmAvFormat_t structure.

Table 6 TXT Output Packet Format

Field	Value
dataClass	avdcGeneric
dataType	avdtGeneric
dataSubtype	avdsGeneric
description	0

Only one buffer is used. The size of this buffers is usually **OUT_DATA_LENGTH_EU_TXT** (42 bytes) or **OUT_DATA_LENGTH_NABTS** (33 bytes). Either way the downstream component has to check the size of the buffer to get the right number of valid data bytes.

L21 Output Packet Format

The Line21 packet has the generic format with a single buffer being used to store the data. The buffer for Line21 sliced data stores just four bytes. The ordering of these bytes is endian dependent. For accessing the data the downstream component has to address the data by indexing the byte organized buffer. data[0] contains "1" if the packet contains valid data, "0" if in-valid. data[1] is "1" if the Line21 data is corresponding to video field two. It is "0" if the Line21 data belongs to field one. data[2] and data[3] contain the two bytes which contain the EIA-608 encoded data.

WSS Output Packet Format

The data size of a full WSS data packet is **OUT_DATA_LENGTH_WSS** (14 bytes). The decoded bytes are stored in the output packet in the same order they have been sent out. First byte on lowest memory address, last byte on highest memory address.

VPS Output Packet Format

The data size of a full VPS data packet is **OUT_DATA_LENGTH_VPS** (26 bytes). The decoded bytes are stored in the output packet in order they have been sent out. First byte on lowest memory address, last byte on highest memory address.

VcapVI Error

Video capturer errors which disturb the normal operation of the component are handled using the error callback function. This is mainly done in the video in ISR. It is up to the application to install such a callback function. These calls are reporting problems coming from the dataout function call or by checking the incoming empty packets. The OS error **TMLIBAPP_QUEUE_EMPTY** is passed to an application provided progress callback function. All other errors are reported by the progress function.

VcapVI Progress

The video capturer reports multiple events to the application if enabled. This reporting can be enabled by setting the default instance field **progressReportFlags** with one or more available progress report flags (or-ed together). One is the cycle count at entering the video-in ISR (**VCAPVI_PROGRESS_ISR_ENTRY**). This value can be used for instance for video-in and video-out synchronization algorithms (software PLL). The other one is the information of a lost frame. This information is produced if the capturer expects an empty packet in the output queue but was not able to get one. In this case the already filled packet will not be sent out and will instead be overridden with new incoming digitized data. The progress codes **VCAPVI_PROGRESS_FIELD1** and **VCAPVI_PROGRESS_FIELD2** indicate what video field was currently captured. The application can distinguish between those events by checking the **progressCode** field of the progress arguments.

Table 7 Available progress events of the video capturer

VCAPVI_PROGRESS_ISR_ENTRY	Reports time stamp of entering the video-in ISR. The cycle count is stored in the description field.
VCAPVI_PROGRESS_LOST_FRAME	Reports a lost frame. This is detected by the ISR in case of an empty dataout empty queue. In this case the video capturer did not get a new empty buffer.
VCAPVI_PROGRESS_FIELD1	Reports if captured field was field one.
VCAPVI_PROGRESS_FIELD2	Reports if captured field was field two.
VCAPVI_PROGRESS_LOST_TXT_PACKET	Reports a lost TXT packet. No empty packet was available in the TXT empty queue.
VCAPVI_PROGRESS_LOST_L21_PACKET	Reports a lost L21 packet. No empty packet was available in the L21 empty queue.
VCAPVI_PROGRESS_LOST_WSS_PACKET	Reports a lost WSS packet. No empty packet was available in the WSS empty queue.
VCAPVI_PROGRESS_LOST_VPS_PACKET	Reports a lost VPS packet. No empty packet was available in the VPS empty queue.

The installation of this callback function is optional.

VcapVI Configuration

The following configuration commands are supported.

Table 8 Configuration Commands

Command	Description
VCAPVI_CONFIG_SET_HORZ_OFFSET	This command sets the horizontal offset of the acquisition window of the video capturer. The value is stored in the parameter field of the ptsaControlArgs_t structure.
VCAPVI_CONFIG_SET_VERT_OFFSET	This command sets the vertical offset of the acquisition window of the video capturer. The value is stored in the parameter field of the ptsaControlArgs_t structure.
VCAPVI_CONFIG_SET_Y_THRESHOLD	This command sets the new threshold line of the video-in unit.
VCAPVI_CONFIG_ENABLE_Y_THRESHOLD	This command enables the threshold feature of the video capturer.

Table 8 Configuration Commands

Command	Description
VCAPVI_CONFIG_DISABLE_Y_THRESHOLD	This command disables the threshold feature of the video capturer. If threshold is disabled the packet is send out at the end of a field/frame.
VCAPVI_CONFIG_GET_HORZ_OFFSET	This command retrieves the current horizontal offset of the video digitizer. The returned value is stored in the parameter field of the ptsaControlArgs_t structure.
VCAPVI_CONFIG_GET_VERT_OFFSET	This command retrieves the current vertical offset of the video digitizer. The returned value is stored in the parameter field of the ptsaControlArgs_t structure.
VCAPVI_CONFIG_STATUS_Y_THRESHOLD	This command retrieves the current status of the threshold feature. If returns **True** if threshold is enabled and **False** if disabled.

VcapVI API Data Structures

This section presents the TriMedia Video Capturer data structures.

Name	Page
tmolVcapVICapabilities_t	124
tmolVcapVIInstanceSetup_t	125

tmolVcapVICapabilities_t

```
typedef struct tmolVcapVICapabilities_t{
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tmolVcapVICapabilities_t, *ptmolVcapVICapabilities_t;
```

Fields

`defaultCapabilities`	Refer to tsa.h.

Description

The video capturer does not provide special capabilities information. Therefore, it simply contains a pointer to the default capabilities.

tmolVcapVIInstanceSetup_t

```
typedef struct tmolVcapVIInstanceSetup_t{
   ptsaDefaultInstanceSetup_t   defaultSetup;
   tmVideoAnalogStandard_t      videoStandard;
   tmVideoAnalogAdapter_t       videoAdapter;
   UInt32                       capSizeFlag;
   UInt32                       startX;
   UInt32                       startY;
   ptmVideoFormat_t             pOutputFormat;
   Bool                         interlaced;
   Bool                         thresholdReachedEnable;
   UInt32                       yThreshold;
   dataService_t                field1Lines[PAL_VBI_LINES_FIELD1];
   dataService_t                field2Lines[PAL_VBI_LINES_FIELD2];
} tmolVcapVIInstanceSetup_t, *ptmolVcapVIInstanceSetup_t;
```

Fields

`defaultSetup`	Refer to *tsa.h*.
`videoStandard`	Either **vasNTSC**, **vasPAL**, or **vasSECAM**. Default is **vasNTSC**.
`adapterType`	Either **vaaCVBS** or **vaaSvideo**. Default is **vaaCVBS**.
`capSizeFlag`	Either **viFULLRES** or **viHALFRES**. Default is **viFULL-RES**.
`startX`	X-offset of acquisition window. Default is zero.
`startY`	Y-offset of acquisition window. Default is start of active video.
`pOutputFormat`	Pointer to video format structure. The video capturer uses the format information for setting up the video-in unit. If this field is Null, the format information is taken from the output descriptor structure. It is recommended to pass an output format to the video capturer. Default is Null.
`interlaced`	True if the capturer has to skip one line to put two consecutive lines of one field in a packet (interlaced organized buffer). **False** to make the capturer putting two consecutive lines right next to each other (plain organized buffer). Default is **True**.
`thresholdReachedEnable`	Enables the threshold feature of the video-in unit. **True** tells the video digizer to send out a half-filled packet at line number **yThreshold**. Default is False.
`yThreshold`	Specifies the line number when a half filled packet has to be send out. If the field **threshold-**

	ReachedEnable is False this value will be ignored. Default is zero.
`field1Lines`	Array of VBI lines. Each can be configured what VBI data has to be sliced in field one. All available constants of type **dataService_t** are allowed. Default is **DT_DO_NOT_ACQUIRE**, which means, no slicing in that particular line.
`field2Lines`	Array of VBI lines. Each can be configured what VBI data has to be sliced in field two. All available constants of type **dataService_t** are allowed. Default is **DT_DO_NOT_ACQUIRE**, which means, no slicing in that particular line.

Description

This structure is used to configure the video capturer. It enables the application to specify parameters such as the video standard, the adaptor type, output format, and data organization. A pointer to the component allocated setup structure can be obtained by the **tmolVcapVIGetInstanceSetup** function. This obtained structure is filled with default values.

VcapVI API Functions

This section presents the functions for the OS Version of the TriMedia Video Capturer API.

tmolVcapVIGetNumberOfUnits

```
extern tmLibappErr_t tmolVcapVIGetNumberOfUnits(
   UInt32   *numberOfUnits
);
```

Parameters

`numberOfUnits`	Pointer (returned) to the supported number of units.

Return Codes

`TMLIBAPP_OK`	Success.

The function can also returns error codes generated by **viGetNumberOfUnits**.

Description

This function returns the number of available hardware units the video capturer supports.

tmolVcapVIGetCapabilities

```
extern tmLibappErr_t tmolVcapVIGetCapabilities(
   ptmolVcapVICapabilities_t   *pCap
);
```

Parameters

`pCap`	Pointer to a variable in which to return a pointer to the video capture capabilities.

Return Codes

`TMLIBAPP_OK`	Success.

The function can also return codes generated by **tmolVcapVIGetCapabilitiesM** and **tmol-VcapVIGetNumberOfUnits**.

Description

This function returns a pointer to the capabilities structure of the default unit (unit zero). This can be used by the format manager to determine if two components can be connected together to form a dataflow.

tmolVcapVIGetCapabilitiesM

```
extern tmLibappErr_t tmolVcapVIGetCapabilitiesM(
   ptmolVcapVICapabilities_t  *pCap,
   unitSelect_t                unitNumber
);
```

Parameters

`pCap`	Pointer to a variable in which to return a pointer to the capabilities data.
`unitNumber`	Unit number to which this instance has to be attached.

Return Codes

`TMLIBAPP_OK`	Success.
`VCAPVI_ERR_VI_NOT_SUPPORTED`	Asserts if the component does not find any video input hardware unit. Usually, this means that the board initialization fails.

The function can also return codes generated by **tmolVcapVIGetNumberOfUnits**, **tsa-BoardGetBoard**, and **viGetCapabilitiesM**.

Description

This function returns pointer to the capabilities structure. This can be used by the format manager to determine if two components can be connected together to form a dataflow.

tmolVcapVIOpen

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

Parameters

`instance`	Pointer to the (returned) instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	The component cannot allocate memory for its instance variables.

The function can also return codes generated by **tmolVcapVIOpenM** and **tmolVcapVIGet-NumberOfUnits**.

Description

This function opens an instance of the video capturer; the default unit is opened (unit zero).

tmolVcapVIOpenM

```
extern tmLibappErr_t tmolVcapVIOpenM(
   Int           *instance,
   unitSelect_t   unitNumber
);
```

Parameters

`instance`	Pointer to the (returned) instance.
`unitNumber`	Number of unit that the video capturer has to drive by creation of an instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MODULE_IN_USE`	The component is already in use.

The function can also return codes generated by **tmolVcapVIGetNumberOfUnits** and **viOpenM**.

Description

This function obtains an instance of the video capturer. The component supports a single instance per hardware unit.

tmolVcapVIClose

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

Parameters

`instance`	Instance value.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the `instance` parameter is an unknown instance.
`TMLIBAPP_ERR_NOT_STOPPED`	Asserts if the instance of video capturer is still running.

The function can also return codes generated by **viStop** and **viClose**.

Description

This function will release the instance.

tmolVcapVIGetInstanceSetup

```
extern tmLibappErr_t tmolVcapVIGetInstanceSetup(
   Int                          instance,
   ptmolVcapVIInstanceSetup_t  *setup
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the **instance** parameter is an unknown instance.

Description

This function returns a pointer to the instance setup structure. The memory for this structure is created automatically when the instance is opened.

tmolVcapVIInstanceSetup

```
extern tmLibappErr_t tmolVcapVIInstanceSetup(
   Int                         instance,
   tmolVcapVIInstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance value.
`setup`	Pointer to the instance setup structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the **instance** parameter is an unknown instance.
`TMLIBAPP_ERR_NULL_IO_DESC`	Asserts if the InOutDescriptor is Null.
`VCAPVI_ERR_SETUP_FORMAT`	Asserts if the content of fields **fieldbased** and **interlaced** of the instance setup structure does not match with the already installed format at the output queue.

The function can also return codes generated by **tmalVcapVIInstanceSetup**.

Description

This function must be called to configure the video capturer. The address of the instance setup structure should be obtained using the **tmolVcapVIGetInstanceSetup** function.

tmolVcapVIStart

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

Parameters

`instance`	Instance value.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the **instance** parameter is an unknown instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the **tmolVcapVIInstanceSetup** function has not been called.
`VCAPVI_ERR_BUFFER_ALLOCATION`	The number of buffers does not match required number of buffers (three).
`VCAPVI_ERR_BUFFER_ALLOCATION`	The memory for buffers is not allocated (Null).
`VCAPVI_ERR_BUFFER_ALIGNMENT`	The pointer to data buffers are not cache aligned (multiple of 64).
`VCAPVI_ERR_BUFFER_SIZE`	The buffer size does not match with image size.
`VCAPVI_ERR_BUFFER_SIZE_ALIGNMENT`	The buffer size is not multiple of cache line size (64).

The function can also return codes generated by **tmalVcapVIStart**.

Description

This function will start the video capturer data streaming operation. It calls the dataoutFunc to obtain an empty packet where captured video data will be stored. The format of this first packet is checked to make sure the packets can hold the data to be captured. The video-in device is then started which will initiate the capture process. After capturing a frame the instance will try to obtain another empty packet. If successful, it will return the current packet containing the captured video data and begin capturing the new packet. If an empty packet is not available, then the progressFunc callback is executed, and the next captured frame will be stored in the current packet. This will overwrite the previous frame.

tmolVcapVIStop

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

Parameters

`instance`	Instance value.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the instance parameter is an unknown instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the **tmolVcapVIInstanceSetup** function has not been called.

The function can also return codes generated by **viStop**.

Description

This function is used to terminate video capture and hence stop data streaming. It will stop the video-in device, and then expel the packet which it was holding. The packet is returned using the dataoutFunc callback.

tmolVcapVIInstanceConfig

```
extern tmLibappErr_t tmolVcapVIInstanceConfig(
   Int                 instance,
   ptsaControlArgs_t   args,
);
```

Parameters

`instance`	Instance value.
`args`	Pointer to a structure specifying how to change the configuration of the running video capturer.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The instance parameter is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	The instance parameter does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	The instance has not been setup using the **tmolVcapVIInstanceSetup** function.

The function can also return codes generated by **tmalVcapVIInstanceConfig**.

Description

This function is used to change the configuration of the video capturer during operation (after the capturer has been started). See *VcapVI Configuration* on page 122 for supported configuration commands.

Chapter 3

Video Digitizer (VdigVI) API

TriMedia Video Digitizer API Overview

The TriMedia video digitizer is an implementation of a TSSA-compliant video input driver. Both AL and OL layers are provided. It delivers data to a downstream TSSA component using either the AL Layer or OL Layer streaming mechanism.

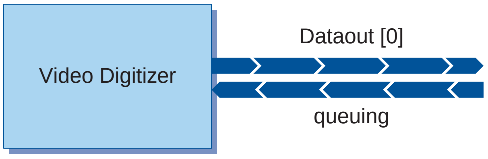


Figure 8 Structure of the Video Digitizer

The video digitizer is a high-level library using the video-in device library that determines on which supported board the TriMedia is mounted. Using the board ID, the device library can control the external video analog/digital converter chip. The application does not have to worry about the required interrupt service routine in order to handle the hardware video-in events—this is covered by the video digitizer. See Figure 2.

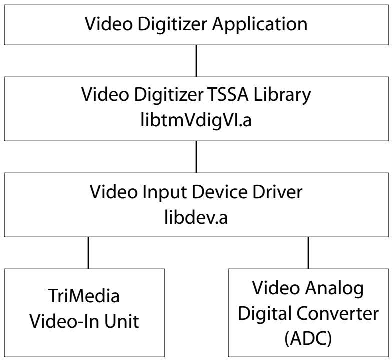

Video Digitizer Application
Video Digitizer TSSA Library
libtmVdigVI.a
Video Input Device Driver
libdev.a
TriMedia
Video-In Unit
Video Analog
Digital Converter
(ADC)

Figure 9 Video Digitizer Architecture

Using the Video Digitizer API

The TriMedia Video Digitizer API is contained within the archived application library libtmVdigVI.a. For using the Video Digitizer AL Layer API, the tmalVdigVI.h header file has to be included; for OL Layer applications, the tmolVdigVI.h header file.

The AL layer

The AL layer supports data streaming operation using the dataoutFunc callback within the video-in interrupt service routine. A typical flow of control is shown in Figure 10.

The application can obtain the capabilities of the component using **tmalVdigVIGet-Capabilities**. This information can be used to determine the supported output formats of the digitizer.

In order to use the VdigVI functionality, an instance of the video digitizer must be created by calling **tmalVdigVIOpen**. If the TriMedia has more than one video-in hardware block the application has to specify which unit has to be driven. **tmalVdigVIOpenM** must be used for this purpose. For instance, the TriMedia family TM-1 only supports one video-in unit, the TM-2 two. Once an instance is installed driving a unit, no more instances of the video digitizer can be created to drive this specific unit.

After the instantiation the application has to initialize the digitizers instance setup structure and call **tmalVdigVIInstanceSetup**. The TSSA streaming model requires a set of callback functions. To make the video digitizer operating in data streaming mode, only the dataoutFunc has to be provided by the application. The video digitizer tries to modify the format information by setting the fields **activeVideoStartX**, **activeVideoStartY**, **activeVideoEndX**, and **activeVideoEndY** of the packet via a provided **progressFunc** callback function. This call is marked with the flag **tsaProgressFlagChangeFormat**. If the application is not interested in the updated values of the format this call can just be returned by with **TMLIBAPP_OK**.

The **tmalVdigVIStart** function begins the data streaming operation. The digitizer will use the **dataoutFunc** callback to obtain an empty packet where the captured video data will be stored. If no empty packet is available the video digitizer does not start with the data streaming. After capturing a image (frame/field), depending on the currently used mode, the digitizer will attempt to acquire another empty packet using the **dataoutFunc** callback. If successful, it will send out the packet with the recently acquired image to the connected downstream component. If acquiring of an empty packet fails, the digitizer will simply use the packet which it has in its possession to store the next image. This overrun condition is signalled by the instance using the **progressFunc** callback. The application must ensure that no additional interrupt depending functions (e.g., **printf**) are being called in that progress function.

As the digitizer is using the video-in peripheral to perform the capture operation, the application is able to perform other operations during this time.

Data streaming can be terminated by calling **tmalVdigVIStop** at any time. This will stop the video-in device, and expel the packets currently being held by the instance. The application can release the instance by calling **tmalVdigVIClose**.

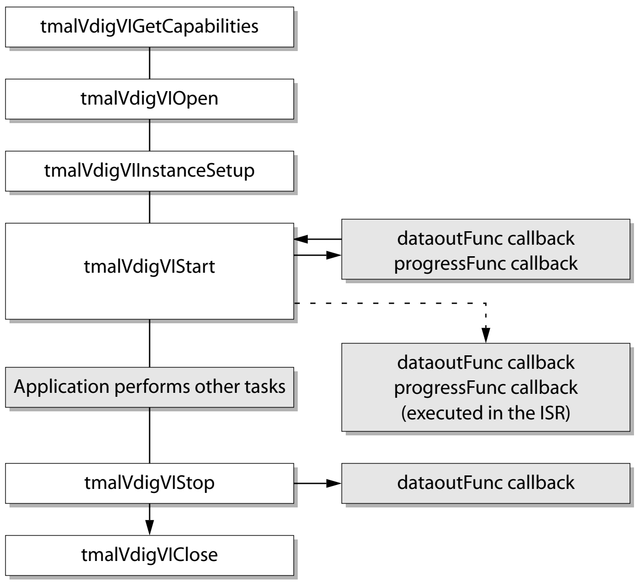


Figure 10 AL Layer data streaming flow control.

The OL Layer

The operating system layer supports data streaming operation using the dataoutFunc callback within the video-in interrupt service routine. A typical flow of control is shown in Figure 11.

First, the application must obtain the capabilities of the component and the hardware unit using **tmolVdigVIGetCapabilities** or **tmolVdigVIGetCapabilitiesM** respectively. The 'M' function has to be used if the application needs to specify an unit other than the default. By default the unit number one will be used if called the **tmolVdigVIGetCapabilities**. The acquired information will be automatically passed to the format manager to ensure that the two components being connected are compatible.

In order to use the VdigVI functionality, an instance of the video digitizer must be created by calling **tmolVdigVIOpen**. If the TriMedia has more than one video-in hardware block the application has to specify which unit has to be driven. **tmolVdigVIOpenM** must be used for this purpose. For instance, the TriMedia family TM-1 only supports one video-in unit., the TM-2 two. Once an instance is installed driving a unit, no more instances of the video digitizer can be created to drive this specific unit. The application should then obtain a pointer to the instance setup structure using **tmolVdigVIGet-InstanceSetup**; this structure is automatically created when the instance is opened. It can then setup the required configuration fields such as the video standard and adapter type. These parameters are passed to the instance by calling **tmolVdigVIInstanceSetup**. Note that by default, the instance will use the dataoutFunc function supplied by the tsaDefaults library. Furthermore, the most of the fields in the setup structure contain default values, accept the field which need to filled by the application. The functional description of the **tmolVdigVIGetInstanceSetup** gives information about the default values.

The **tmolVdigVIStart** function begins the data streaming operation by calling the **tmolVdigVIStart** function. See *The AL layer* on page 141.

Data streaming can be terminated by calling **tmolVdigVIStop** at any time. This will stop the video-in device, and expel the packets currently being held by the instance. The application can release the instance by calling **tmolVdigVIClose**.

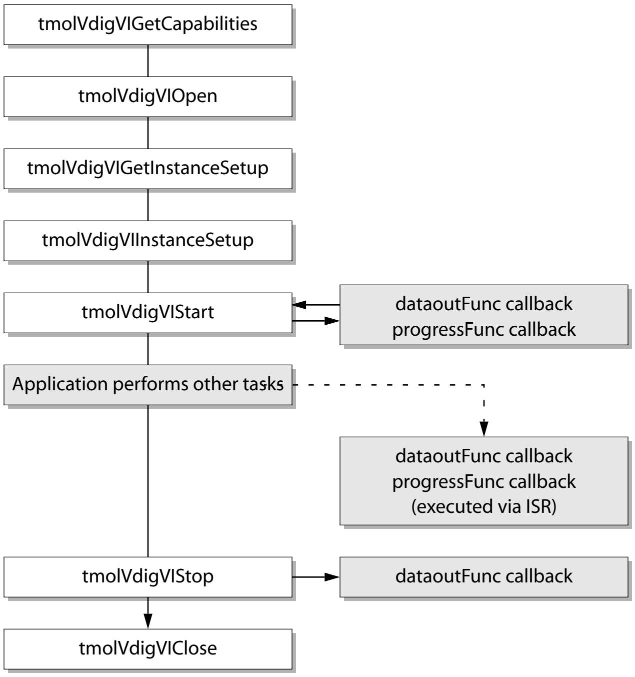


Figure 11 OL Layer data streaming flow control

Line counting issues

To handle the vertical blanking interval (VBI) data in a generic way, that is, having the VBI data always at the beginning of a video packet, the following line counting issues are addressed. The 60 and 50 fields per second video systems use different line counting bases, the 50 Hz system for instance starts counting on the serration pulses and the 60 Hz systems starts counting at the first equalization pulses. The equalization and serration pulses can be seen in the vertical timing diagrams Figure 6 and Figure 7. Those pulses have twice the line frequency and are located in the vertical blanking interval.

Table 9 Field interval definitions according to CCIR 656

Field	SAV/EAV bit	50 Hz (625 lines)	60 Hz (525 lines)
V-digital field blanking			
Field 1	Start (V=1)	Line 624	Line 1
	Finish (V=0)	Line 23	Line 10
Field 2	Start (V=1)	Line 311	Line 264
	Finish (V=0)	Line 336	Line 273
F-digital field identification			
Field 1	F = 0	Line 1	Line 4
Field 2	F = 1	Line 313	Line 266

In 50 Hz systems the **startY**[1] field has an allowed lowest value of -2, to get the content of the first equalization pulse group, since 50 Hz line counting starts after this first equalization pulse group, and VdigVI always delivers the VBI data at the beginning of the video buffer. Since 60Hz counting starts at the beginning of the first equalization pulses, the lowest allowed value in this system is zero. That makes the use of the digitizer component as easy as possible. e.g. an application wants to get line 21 data additional to the active video, the **startY** value has to be set to 21 in either frequency (50 or 60 Hz).

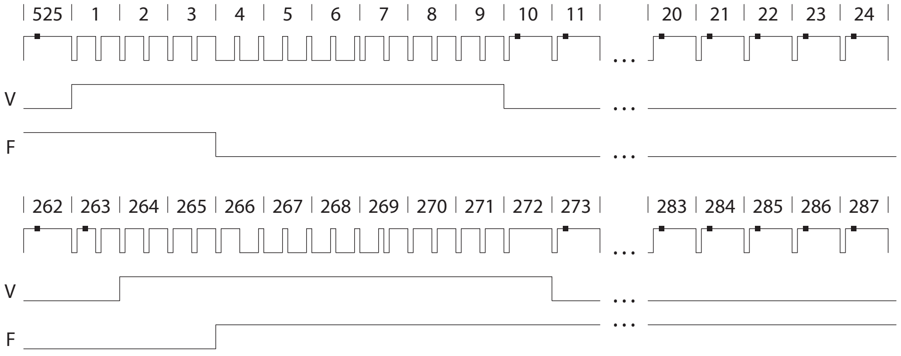


Figure 12 Vertical timing diagram for 60 Hz and its corresponding SAV/SEAV bits.
V: vertical sync. F: field flag

1. startY is defined in the databook in the VI chapter, it is used to vertically position the capture window.

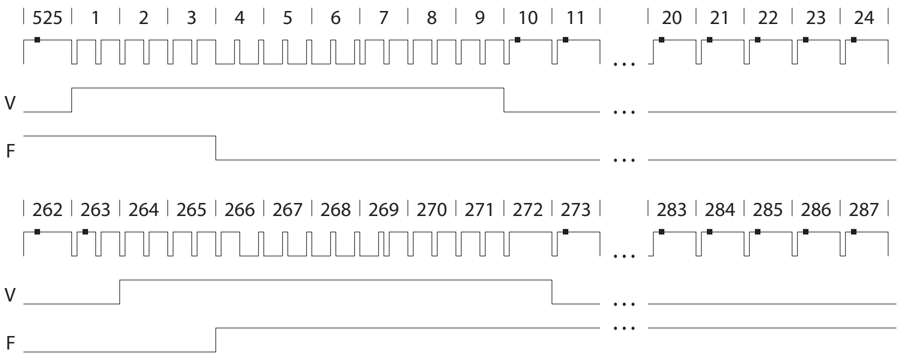


Figure 13 Vertical timing diagram for 50 Hz and its corresponding SAV/EAV bits. V: vertical sync. F: field flag

The video-in hardware block starts counting lines when the V bit in the SAV/EAV codes goes from one to zero. The CCIR-656 standard defines the following lines where this happens: 50 Hz: line 23 and 60 Hz line 10. This leads to the following observation. If the digitizer has to capture before these lines (always the case for VBI capturing in 50 Hz systems), a much higher line count has to be put in the hardware registers to start capturing in the VBI interval. The first field has one line more than the second field (313/312 in 625-line systems and 263/262 in 525-line systems). This leads to the restriction that the resulting line (number in the register) can not be 313 or 263 in 50 Hz or 60 Hz systems respectively, because the internal counter of the VI-block only gets reached in the first field, but never in the second field. This fact causes one little restriction. The internal start value of capturing never can be 313/263. But if an application wants to get the content of this line it has to start with the capturing one line before.

The fields **activeVideoStartX**, **activeVideoStartY**, **activeVideoEndX**, and **activeVideoEndY** have nothing to do with the defined line counting in the 60 Hz and 50 Hz systems. Those values only represents offset values where a downstream component can find the active video area by taking into account how the video data are organized in the video buffer. In interlaced systems the values (times stride) directly lead to the address of the active video data. In field-in -ield systems the content of the structure fields have to be divided by two in order to apply the same calculation like in the interlaced case to get to the right active video addresses. That is why those field have a range from 0 to 576 or from 0 to 480 respectively.

Cache Coherency

When using the OL Layer of the video digitizer, the application just has to use the **tsaIO-DescSetupFlagInvalidateDataout** flag for creation of the data queue between video digitizer and the downstream component. In this case all cache coherency issues are automatically handled by the tsaDefaults library.

When the application is using the video digitizer AL Layer, it must consider cache coherency issues. For example, if the DSPCPU will read the captured video data, then the application must perform a cache invalidate operation on the video data before the DSPCPU accesses this part of the memory. This can simply be performed using the **_cache_invalidate** function on the relevant video data. Cache aligned video buffers can easily be created and destroyed using the **_cache_malloc** and **_cache_free** functions respectively. After the application has allocated the memory for the video buffers, it must perform a **_cache_copyback** operation for each buffer. This only needs to be performed on buffer creation and ensures that the memory has been flushed out of the DSPCPU data cache.

TriMedia Video Digitizer Inputs and Outputs

The video digitizer is a data source therefore it has no input pin and provides only one output pin. The output format can be specified using the instance setup function. The field **pOutputFormat** has to be filled by the application. The value must be a pointer to a **tmVideoFormat_t** structure. For easier bug tracking, the video digitizer throws an assertion failure if the installed format does not match with the format installed on the output queue. The video digitizer checks if the containing parameters are supported by the library. Currently only the TV standards NTSC and PAL and its related parameters are supported.

In the AL layer the dataout callback has to be provided by the application to make the output pin working. The instance setup function returns with an error code if a Null pointer is passed to the library. The dataout callback is called in the video in interrupt context. This means that the application-supplied callback function should execute as fast as possible so the time spent inside the interrupt routine be kept to a minimum.

In the OL layer, if the **dataoutFunc** field of the default setup structure is Null, the default dataout callback function provided by the tsaDefaults library will be used.

Packet Formats

The video digitizer uses the standard packet data types defined in the **tmAvFormats.h** include file. The output uses the **tmAvPacket_t** structure to specify the packet. The captured YUV data is stored in three buffers, with the Y data contained in **buffer[0]**, and the UV data contained in **buffer[1]** and **buffer[2]** respectively.

Each packet contains a header structure providing information concerning the packet data. The format field is a pointer to a **tmVideoFormat_t** structure which specifies the format and the image size. There are restrictions on the type of video formats that can be used by the video digitizer. These will be described next.

The main image output packet can be either **vdfYUV422Planar** or **vdfYUV422Interspersed**. No YUV420 format is supported by the video-in unit. If the video digitizer is used also for capturing of vertical blanking interval data, the vdfYUV422Planar flags needs to be used, because in this mode the video in hardware does not perform any filtering of the incoming data. The **pOutputFormat** field in the instance setup structure should be initialized with the following values:

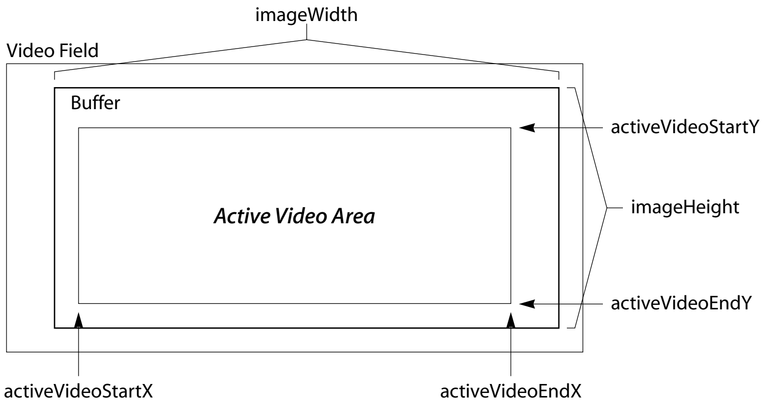
Field	**Set by**	**Value**
dataClass	App	avdcVideo
dataType	App	vtfYUV
dataSubtype	App	vdfYUV422Planar or vdfYUV422Interspersed
description	App	vdfInterlaced, vdfFieldInFrame, or vdfFieldInField
imageWidth	App	Width of video frame in pixels (luminance)
imageHeight	App	Height of video frame in lines (luminance)
imageStride	App	Stride of video frame in bytes (luminance)
activeVideoStartX	VdigVI	Defines pixel offset in horizontal direction from start of video buffer to beginning of active video.
activeVideoStartY	VdigVI	Defines number of lines from start of video buffer to beginning of active video
activeVideoEndX	VdigVI	Defines end of active video area in number of pixels within the video buffer. It is an absolute position.
activeVideoEndY	VdigVI	Defines end of active video area in number of lines within the video buffer. It is an absolute position.
videoStandard	VdigVI	Defines analog video standard that served as source of digitized image

The **description** field of the format structure is set by the video digitizer automatically depending on the instance setup fields **fieldBased** and **interlaced**, but it still checks if the resulting description matches with the previously installed format of the output queue. Four different combinations are possible. See Table 10.

Table 10 The description field is set by the VdigVI library during the instance setup

description	fieldBased == True	fieldBased == False
interlaced == True	**vdfFieldInFrame** The first field is stored just as in the vdfInterlaced mode. The second field is stored just as in the vdfInterlaced mode, but not in the same packet. Sent packets alternately have the first field filled (second field not updated), and second field filled (first field not updated).	Default **vdfInterlaced** A complete frame is written in the packet buffer. The fields are stored interlaced. The packets are sent out frame based.
interlaced == False	**vdfFieldInField** Only one field is written in the packet buffer. No space is between two consecutive lines of one field. The packet is sent out field based.	**vdfFieldInField** The previous captured field is overridden by the current field. The packets are sent out frame based.

In field-based operation, every packet is marked with the field type using the flags field of the header structure. The second field will have the **avhField2** bit set, whereas the first field will have this bit clear. Therefore, a downstream component can interrogate the packets description and field information to determine which data bytes in the packet are valid. It is possible to build up a complete frame by sending a half-filled frame back to the digitizer, which will insert the missing field into the correct memory locations of the packet buffer.



The fields **activeVideoStartX**, **activeVideoStartY**, **activeVideoEndX**, **activeVideoEndY**, and **videoStandard** are also set by the video digitizer automatically. In this case it does not check if it matches with the current queue setup. Depending what analog video standard was chosen the video digitizer sets these fields accordingly. Using the location of the active video area a downstream component has access to additional information, such as VBI inserted data, which is transmitted by the video signal in parallel.

TriMedia Video Digitizer Error

Video digitizer errors which would disturb the operation of the component are handled using the error callback function. The callback function is usually called from within the video-in ISR. The application is responsible for supplying a function to handle these error conditions. Error conditions which arise include problems whereas calling the dataout function and the validity of incoming empty packets. Underrun errors which occur when the component fails to obtain a new empty packet are reported through the progress callback function. Progress functions are described in the next section.

TriMedia Video Digitizer Progress

The video digitizer reports multiple events to the application if enabled. This reporting can be enabled by setting the default instance field **progressReportFlags** with OR'd available progress report flags. One is the cycle count at entering the video-in ISR (**VDIGVI_PROGRESS_ISR_ENTRY**). This value can be used for instance for video-in and video-out synchronization algorithms. The other one is the information of a lost frame. This information is produced if the digitizer expects an empty packet in the output queue but was not able to get one. In this case the already filled packet will not be sent out and will instead be overridden with new incoming digitized data. The progress codes **VDIGVI_PROGRESS_FIELD1** and **VDIGVI_PROGRESS_FIELD2** indicate what video field was currently captured. The application can distinguish between those events by checking the **progressCode** field of the progress arguments.

Table 11 Available progress events of the video digitizer

VDIGVI_PROGRESS_ISR_ENTRY	Reports time stamp of entering the video-in ISR. The cycle count is stored in the description field.
VDIGVI_PROGRESS_LOST_FRAME	Reports a lost frame. This is detected by the ISR in case of an empty dataout empty queue. In this case the video digitizer did not get a new empty buffer.
VDIGVI_PROGRESS_FIELD1	Reports if captured field was field one.
VDIGVI_PROGRESS_FIELD2	Reports if captured field was field two.

The installation of this callback function is optional.

Video Digitizer API Data Structures

This section presents the TriMedia Video Digitizer data structures.

Name	Page
tmolVcapVICapabilities_t	124
tmolVcapVIInstanceSetup_t	125

tmalVdigVICapabilities_t, tmolVdigVICapabilities_t

```
typedef struct tmalVdigVICapabilities_t{
   ptsaDefaultCapabilities_t   defaultCapabilities;
   pviCapabilities_t           viCapabilities;
} tmalVdigVICapabilities_t, *ptmalVdigVICapabilities_t;
typedef tmalVdigVICapabilities_t tmolVdigVICapabilities_t;
typedef ptmalVdigVICapabilities_t ptmolVdigVICapabilities_t;
```

Fields

`defaultCapabilities`	Refer to tsa.h.
`viCapabilities`	Pointer to the video-in device library capabilities.

Description

This structure stores the capabilities of the video digitizer.

tmalVdigVIInstanceSetup_t, tmolVdigVIInstanceSetup_t

```
typedef struct tmalVdigVIInstanceSetup_t{
   ptsaDefaultInstanceSetup_t  defaultSetup;
   tmVideoAnalogStandard_t     videoStandard;
   tmVideoAnalogAdapter_t      videoAdapter;
   UInt32                      capSizeFlag;
   UInt32                      startX;
   UInt32                      startY;
   ptmVideoFormat_t            pOutputFormat;
   Bool                        fieldBased;
   Bool                        interlaced;
   Bool                        thresholdReachedEnable;
   UInt32                      yThreshold;
   Bool                        startYisScanLineNumber;
} tmalVdigVIInstanceSetup_t, *ptmalVdigVIInstanceSetup_t;
typedef tmalVdigVIInstanceSetup_t tmolVdigVIInstanceSetup_t;
typedef ptmalVdigVIInstanceSetup_t ptmolVdigVIInstanceSetup_t;
```

Fields

`defaultSetup`	Refer to *tsa.h*.
`videoStandard`	Either **vasNTSC**, **vasPAL**, or **vasSECAM**. Default is **vasNTSC**.
`adapterType`	Either **vaaCVBS** or **vaaSvideo**. Default is **vaaCVBS**.
`capSizeFlag`	Either **viFULLRES** or **viHALFRES**. Default is **viFULLRES**.
`startX`	x-offset of acquisition window. Default is zero.
`startY`	y-offset of acquisition window. Default is start of active video.
`pOutputFormat`	Pointer to video format structure. The video digitizer uses the format information for setting up the video in unit. If this field is Null, the format information is taken from the output descriptor structure. It is recommended to pass an output format to the video digitizer. Default is Null.
`fieldBased`	True if the digitizer has to send captured packets in a fieldbased frequency. False to make digitizer sending packets in a frame based frequency. Default is False.
`interlaced`	**True** if the digitizer has to skip one line to put two consecutive lines of one field in a packet (interlaced organized buffer). False to make the digitizer putting two consecutive lines right next to each other (plain organized buffer). Default is True.

`thresholdReachedEnable`	Enables the threshold feature of the video-in unit. **True** tells the video digizer to send out a half-filled packet at line number **yThreshold**. Default is False.
`yThreshold`	Specifies the line number when a half filled packet has to be send out. If the field **threshold-ReachedEnable** is False this value will be ignored. Default is zero.
`startYisScanLineNumber`	Used for VBI support.

Description

This structure is used to configure the video digitizer. It enables the application to specify parameters such as the video standard, the adaptor type, output format, packet output frequency, and data organization. In OL layer a pointer to the component allocated setup structure can be get by the get instance setup function. This obtained structure is also filled with default values. All the mentioned default values are only available in the OL layer by calling **tmolVdigVIGetInstanceSetup**.

AL Layer Video Digitizer API Functions

This section presents the TriMedia Video Digitizer API functions.

Name	Page
tmalVdigVIGetNumberOfUnits	155
tmalVdigVIGetCapabilities	156
tmalVdigVIGetCapabilitiesM	157
tmalVdigVIOpen	158
tmalVdigVIOpenM	159
tmalVdigVIClose	160
tmalVdigVIInstanceSetup	161
tmalVdigVIStart	162
tmalVdigVIStop	163
tmalVdigVIInstanceConfig	164

tmalVdigVIGetNumberOfUnits

```
extern tmLibappErr_t tmalVdigVIGetNumberOfUnits(
   UInt32   *numberOfUnits
);
```

Parameters

`numberofUnits`	Pointer to integer that contains the supported number of units after the function has returned successfully.

Return Codes

`TMLIBAPP_OK`	Success.

The function can also return codes from **viGetNumberOfUnits**.

Description

This function returns the number of available hardware units the video digitizer supports.

tmalVdigVIGetCapabilities

```
extern tmLibappErr_t tmalVdigVIGetCapabilities(
   ptmalVdigVICapabilities_t   *pCap
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.

The function can also return codes from **tmalVdigVIGetCapabilitiesM**.

Description

This function returns a pointer to the video digitizer capabilities such as version information. It simply calls the **tmalVdigVIGetCapabilitiesM** function with the default unit number (**unit0**).

tmalVdigVIGetCapabilitiesM

```
extern tmLibappErr_t tmalVdigVIGetCapabilitiesM(
   ptmalVdigVICapabilities_t   *pCap,
   unitSelect_t                 unitNumber
);
```

Parameters

`pCap`	Pointer to a variable in which to return a pointer to capabilities data.
`unitNumber`	Unit number where the capabilities have to be retrieved from.

Return Codes

`TMLIBAPP_OK`	Success.
`VDIGVI_ERR_VI_NOT_SUPPORTED`	Can assert if the component can’t find any video input hardware unit.

The function can also return codes from **tmalVdigVIGetNumberOfUnits**, **tsaBoardGetBoard**, **viGetNumberOfUnits**, and **viGetCapabilitiesM**.

Description

This function returns a pointer to the video digitizer capabilities.

tmalVdigVIOpen

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

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.

The function can also return codes from **tmalVdigVIOpenM**.

Description

This function is used to obtain an instance of the video digitizer. The component supports a single instance per hardware unit. This function creates an instance from **unit0**. If an application needs to use a different unit it should use the **tmalVdigVIOpenM** function.

tmalVdigVIOpenM

```
extern tmLibappErr_t tmalVdigVIOpenM(
   Int           *instance,
   unitSelect_t   unitNumber
);
```

Parameters

`instance`	The instance.
`unitNumber`	Number of the video-in unit the application wishes to use with the video digitizer.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MODULE_IN_USE`	The component is already being used.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	No memory is available to store capabilities of component.

The function can also return codes from **tmalVdigVIGetNumberOfUnits** and **viOpenM**.

Description

This function is used to obtain an instance of the video digitizer. The component supports a single instance per hardware unit.

tmalVdigVIClose

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

Parameters

`instance`	Instance value.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the **instance** parameter is Null or does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_STOPPED`	Asserts if the **instance** has not been stopped.

The function can also return codes from **viStop** and **viClose**.

Description

This function will release an instance.

tmalVdigVIInstanceSetup

```
extern tmLibappErr_t tmalVdigVIInstanceSetup(
   Int                           instance,
   tmalVdigVIInstanceSetup_t    *setup
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The **instance** parameter is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	The **instance** parameter does not match the currently opened instance.
`TMLIBAPP_ERR_INVALID_SETUP`	Asserts if the **defaultSetup** pointer contained within the setup structure is Null
`TMLIBAPP_ERR_NULL_DATAOUT_FUNC`	Asserts if the dataoutFunc callback is Null.
`VDIGVI_ERR_SETUP_FORMAT`	Asserts if the content of fields **fieldbased** and **interlaced** of the instance setup structure does not match with the already installed format at the output queue.

Description

This function configures the digitizer. It is important to ensure that the application specifies a dataoutFunc callback which will be used to obtain and release packets.

tmalVdigVIStart

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

Parameters

`instance`	Instance value.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if **the instance** parameter is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	Asserts if the **instance** parameter does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the instance has not been setup using the `tmalVdigVIInstanceSetup` function.
`VDIGVI_ERR_BUFFER_ALLOCATION`	The number of buffers does not match required number of buffers (three).
`VDIGVI_ERR_BUFFER_ALLOCATION`	Memory for buffers is not allocated (Null).
`VDIGVI_ERR_BUFFER_ALIGNMENT`	Data buffers are not cache aligned (multiple of 64).
`VDIGVI_ERR_BUFFER_SIZE`	Buffer size does not match with image size.
`VDIGVI_ERR_BUFFER_SIZE_ALIGNMENT`	Buffer size is not multiple of cache line size (64).

The function can return codes from **viYUVSetup** and **viStart**.

Description

This function will start the video digitizer data streaming operation. It calls the dataoutFunc to obtain an empty packet where captured video data will be stored. The format of this first packet is checked to make sure the packets can hold the to be captured data. The video-in device is then started which will initiate the capture process. After capturing a frame the instance will try to obtain another empty packet. If successful it will return the current packet containing the captured video data and begin capturing to the new packet. If an empty packet is not available, then the progressFunc callback is executed, and the next captured frame will be stored in the current packet. This will overwrite the previous frame. The application must provide its own dataout callback function which provides functionality to get empty packets and return full packets. The address of this function must be specified during instance setup.

tmalVdigVIStop

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

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The **instance** parameter is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	The **instance** parameter does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	The instance has not been setup using the **tmalVdigVIInstanceSetup** function.

The function can return codes from **viStop**.

Description

This function is used to terminate video capture and hence stop data streaming. It will stop the video-in device, and then expel the packet which it was holding. The packet is returned using the dataoutFunc callback.

tmalVdigVIInstanceConfig

```
extern tmLibappErr_t tmalVdigVIInstanceConfig(
   Int                 instance,
   ptsaControlArgs_t   args
);
```

Parameters

`instance`	The instance.
`args`	Pointer to a structure specifying how to change the configuration of the running video digitizer.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The **instance** parameter is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	The **instance** parameter does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	The instance has not been setup using the **tmalVdigVIInstanceSetup** function.
`VDIGVI_ERR_CONFIG_UNKNOWN_COMMAND`	Asserts if config function gets not supported command codes.

Description

This function is used to change the configuration of the video digitizer. The following constants at the command field of the ptsaControlArgs_t are supported.

Configuration Commands

`VDIGVI_CONFIG_SET_HORZ_OFFSET`	This command sets the horizontal offset of the acquisition window of the video digitizer. The value is stored in the parameter field of the ptsaControlArgs_t structure.
`VDIGVI_CONFIG_SET_VERT_OFFSET`	This command sets the vertical offset of the acquisition window of the video digitizer. The value is stored in the parameter field of the ptsaControlArgs_t structure.
`VDIGVI_CONFIG_SET_Y_THRESHOLD`	This command sets the new threshold line of the video-in unit.
`VDIGVI_CONFIG_ENABLE_Y_THRESHOLD`	This command enables the threshold feature of the video digitizer.

`VDIGVI_CONFIG_DISABLE_Y_THRESHOLD`	This command disables the threshold feature of the video digitizer. If threshold is disabled the packet is send out at the end of a field/frame.
`VDIGVI_CONFIG_GET_HORZ_OFFSET`	This command retrieves the current horizontal offset of the video digitzer. The returned value is stored in the parameter field of the **ptsaControlArgs_t** structure.
`VDIGVI_CONFIG_GET_VERT_OFFSET`	This command retrieves the current vertical offset of the video digitzer. The returned value is stored in the parameter field of the **ptsaControlArgs_t** structure.
`VDIGVI_CONFIG_STATUS_Y_THRESHOLD`	This command retrieves the current status of the threshold feature. If returns True if threshold is enabled and False if disabled.

OL Layer Video Digitizer API Functions

This section presents the functions for the OS Version of the Video Digitizer API.

Name	Page
tmolVcapVIGetNumberOfUnits	128
tmolVcapVIGetCapabilities	129
tmolVcapVIGetCapabilitiesM	130
tmolVcapVIOpen	131
tmolVcapVIOpenM	132
tmolVcapVIClose	133
tmolVcapVIGetInstanceSetup	134
tmolVcapVIInstanceSetup	135
tmolVcapVIStart	136
tmolVcapVIStop	137
tmolVcapVIInstanceConfig	138

tmolVdigVIGetNumberOfUnits

```
extern tmLibappErr_t tmolVdigVIGetNumberOfUnits(
   UInt32   *numberOfUnits
);
```

Parameters

`numberofUnits`	Pointer to a variable in which to return the supported number of units.

Return Codes

`TMLIBAPP_OK`	Success.

The function can also return codes from **tmalGetNumberOfUnits**.

Description

This function returns the number of available hardware units the video digitizer supports.

tmolVdigVIGetCapabilities

```
extern tmLibappErr_t tmolVdigVIGetCapabilities(
   ptmolVdigVICapabilities_t   *pCap
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.

The function can also return codes from **tmolVdigVIGetCapabilitiesM**.

Description

This function returns pointer to the capabilities structure. This can be used by the format manager to determine if two components can be connected together to form a dataflow.

The function simply calls the **tmolVdigVIGetCapabilitiesM** function with the default unit number (**unit0**).

tmolVdigVIGetCapabilitiesM

```
extern tmLibappErr_t tmolVdigVIGetCapabilitiesM(
   ptmolVdigVICapabilities_t   *pCap,
   unitSelect_t                 unitNumber
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`VDIGVI_ERR_VI_NOT_SUPPORTED`	Can assert if the component does not find any video input hardware unit.

The function can also return codes from **tmalVdigVIGetCapabilitiesM**.

Description

This function returns a pointer to the capabilities structure. It can be used by the format manager to determine if two components can be connected together to form a dataflow.

tmolVdigVIOpen

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

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.

or return codes of the internally called **tmolVdigVIOpenM** function.

Description

This opens an OL Layer instance of the video digitizer. The default unit is opened (**unit0**). If an application wants to use a different unit, it should use the **tmalVdigVIOpenM** function.

tmolVdigVIOpenM

```
extern tmLibappErr_t tmolVdigVIOpenM(
   Int           *instance,
   unitSelect_t   unitNumber
);
```

Parameters

`instance`	The instance.
`unitNumber`	Number of the video-in unit the applications wishes to use with the video digitizer.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MODULE_IN_USE`	The component is already being used.

The function can also return codes from **tmalVdigVIOpenM**.

Description

This function is used to obtain an instance of the video digitizer. The component supports a single instance per hardware unit.

tmolVdigVIClose

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

Parameters

`instance`	Instance value.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the `instance` parameter is an unknown instance.
`TMLIBAPP_ERR_NOT_STOPPED`	Asserts if the instance of video digitizer is still running.

The function can also return codes from **tmalVdigVIClose**.

Description

This function will release the instance.

tmolVdigVIGetInstanceSetup

```
extern tmLibappErr_t tmolVdigVIGetInstanceSetup(
   Int                            instance,
   ptmolVdigVIInstanceSetup_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`	Can assert if the **instance** parameter is an unknown instance.

Description

This returns a pointer to the OL Layer instance setup structure. The memory for this structure is created automatically when the instance is opened.

tmolVdigVIInstanceSetup

```
extern tmLibappErr_t tmolVdigVIInstanceSetup(
   Int                         instance,
   tmolVdigVIInstanceSetup_t  *setup
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Can assert if the **instance** parameter is an unknown instance.
`TMLIBAPP_ERR_NULL_IO_DESC`	Can assert if the InOutDescriptor is Null.
`VDIGVI_ERR_SETUP_FORMAT`	Can assert if the content of fields **fieldbased** and **interlaced** of the instance setup structure does not match with the already installed format at the output queue.

The function can also return codes from **tmalVdigVIInstanceSetup**.

Description

This function must be called to configure the video digitizer. The address of the instance setup structure should be obtained using the **tmolVdigVIGetInstanceSetup** function.

tmolVdigVIStart

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

Parameters

`instance`	The instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the **instance** parameter is an unknown instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the **tmolVdigVIInstanceSetup** function has not been called.
`VDIGVI_ERR_BUFFER_ALLOCATION`	The number of buffers does not match required number of buffers (three).
`VDIGVI_ERR_BUFFER_ALLOCATION`	The memory for buffers is not allocated (Null).
`VDIGVI_ERR_BUFFER_ALIGNMENT`	The pointer to data buffers are not cache aligned (multiple of 64).
`VDIGVI_ERR_BUFFER_SIZE`	The buffer size does not match with image size.
`VDIGVI_ERR_BUFFER_SIZE_ALIGNMENT`	The buffer size is not multiple of cache line size (64).

The function can also return codes from **tmalVdigVIStart**.

Description

This function will start the video digitizer data streaming operation. It calls the dataoutFunc to obtain an empty packet where captured video data will be stored. The format of this first packet is checked to make sure the packets can hold the to be captured data. The video-in device is then started which will initiate the capture process. After capturing a frame the instance will try to obtain another empty packet. If successful it will return the current packet containing the captured video data and begin capturing to the new packet. If an empty packet is not available, then the progressFunc callback is executed, and the next captured frame will be stored in the current packet. This will overwrite the previous frame. By default the video digitizer will use the dataout callback function provided with the tsaDefaults library.

tmolVdigVIStop

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

Parameters

`instance`	Instance value.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the **instance** parameter is an unknown instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the **tmolVdigVIInstanceSetup** function has not been called.

The function can also return codes from **tmalVdigVIStop**.

Description

This function is used to terminate video capture and hence stop data streaming. It will stop the video-in device, and then expel the packet which it was holding. The packet is returned using the dataoutFunc callback.

tmolVdigVIInstanceConfig

```
extern tmLibappErr_t tmolVdigVIInstanceConfig(
   Int                 instance,
   ptsaControlArgs_t   args,
);
```

Parameters

`instance`	The instance.
`args`	Pointer to a structure specifying how to change the configuration of the running video digitizer.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	The **instance** parameter is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	The **instance** parameter does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	The instance has not been setup using the **tmolVdigVIInstanceSetup** function.

The function can also return codes from **tmalVdigVIInstanceConfig**.

Description

This function is used to change the configuration of the video digitizer during operation (after the digitizer has been started). The following constants at the command field of the **ptsaControlArgs_t** are supported.

Configuration Commands

`VDIGVI_CONFIG_SET_HORZ_OFFSET`	This command sets the horizontal offset of the acquisition window of the video digitizer. The value is stored in the parameter field of the **ptsaControlArgs_t** structure.
`VDIGVI_CONFIG_SET_VERT_OFFSET`	This command sets the vertical offset of the acquisition window of the video digitizer. The value is stored in the parameter field of the **ptsaControlArgs_t** structure.
`VDIGVI_CONFIG_SET_Y_THRESHOLD`	This command sets the new threshold line of the video-in unit.
`VDIGVI_CONFIG_ENABLE_Y_THRESHOLD`	This command enables the threshold feature of the video digitizer.

`VDIGVI_CONFIG_DISABLE_Y_THRESHOLD`	This command disables the threshold feature of the video digitizer. If threshold is disabled the packet is send out at the end of a field/frame.
`VDIGVI_CONFIG_GET_HORZ_OFFSET`	This command retrieves the current horizontal offset of the video digitzer. The returned value is stored in the parameter field of the **ptsaControlArgs_t** structure.
`VDIGVI_CONFIG_GET_VERT_OFFSET`	This command retrieves the current vertical offset of the video digitzer. The returned value is stored in the parameter field of the **ptsaControlArgs_t** structure.
`VDIGVI_CONFIG_STATUS_Y_THRESHOLD`	This command retrieves the current status of the threshold feature. If returns **True** if threshold is enabled and **False** if disabled.

Chapter 4

Video Renderer (VrendVO) API

Video Renderer API Overview

The TriMedia video renderer is an implementation of a video output driver which complies with the TriMedia streaming architecture specification.

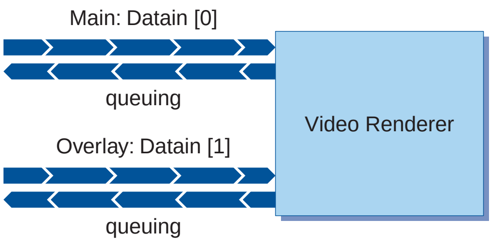


Figure 14 Structure of the Video Renderer

The video renderer accepts data from an application, using either a non-streaming or streaming interface. Both AL and OL layers are provided, as indicated in Figure 15.

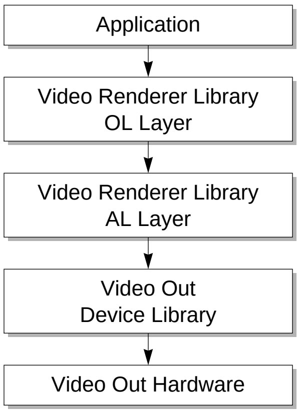


Figure 15 Video Renderer Architecture

Using the Video Renderer API

The TriMedia Video Renderer API is contained within the archived application library libtmVrendVO.a. To use the Video Renderer AL layer API, you must include the tmalVrendVO.h header file; for OL layer applications you must include the tmolVrendVO.h header file.

The AL Layer

The operating system independent layer supports both non-streaming and streaming operation.

In non-data streaming mode, the application explicitly calls the **tmalVrendVORender-Frame** function to transfer the frame to the video renderer instance. A diagram of the typical control flow is shown in Figure 16.

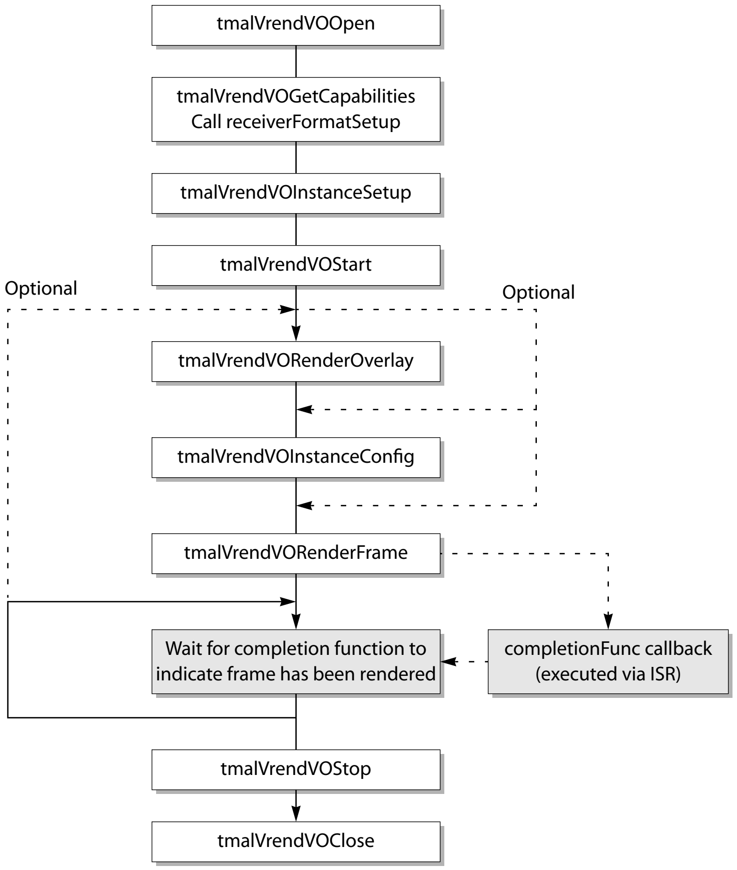


Figure 16 AL Layer Non-Data Streaming Flow Control

An instance of the video renderer must first be created by calling the **tmalVrendVOOpen** function; the component only allows one instance to be open at any moment of time. Once opened, the application should obtain the capabilities of the renderer using **tmalVrendVOGetCapabilities**. It should then call the video renderers **receiverFormatSetup** callback function to specify the output format of the instance; this configures the output height, width, and stride.

The instance should then be setup by initializing the **tmalVrendVOInstanceSetup_t** structure and calling **tmalVrendVOInstanceSetup**. Parameters which may be setup include the video standard, overlay enable, and application specific completion function. Note that for non-data streaming the datainFunc must be set to Null.

The video renderer can then be started using **tmalVrendVOStart**; this informs the renderer to expect data and consequently, to log underrun errors if data is not present. The application may then call **tmalVrendVORenderFrame** to display a frame. The renderer is able to queue up to four frames for display. Once the renderer has displayed a frame, it will call the completion function, and pass the packet ID as the flags argument; the application can use this to determine when a frame has been displayed. Note that if the instance currently has only one frame in it's queue, then this frame will be displayed repeatedly. In this case, the completion function is only called once another frame has been passed to it.

The **tmalVrendVORenderOverlay** function should be used to pass an overlay image to the renderer. The application may call this repeatedly to render new overlays on the main image. The **tmalVrendVOInstanceConfig** can be called to change instance parameters such as the main image position, and whether the overlay should be displayed.

By calling **tmalVrendVOStop**, the renderer will stop displaying images, and return any packets that are stored in it's internal queue. The completion function will be called for each packet on the queue, with the completion function flags argument being set to the relevant packet ID. Finally, when the application calls **tmalVrendVOClose**, the instance will be freed, enabling another task to use the renderer.

In the AL layer streaming operation, the video renderer uses the datainFunc callback within the video interrupt service routine to obtain video packets to be displayed. A typical flow of control is shown in Figure 17.

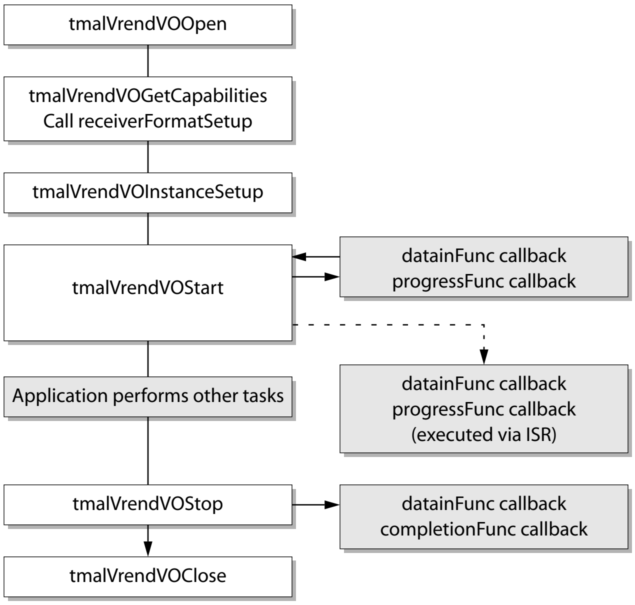


Figure 17 AL Layer Data Streaming Flow Control

A video renderer instance is opened and it’s output configuration is initialized in identical fashion to the non-streaming flow described previously. In streaming mode, the application must set the datainFunc callback in the instance setup structure to point to an application supplied datain callback function. This will be used by the component to obtain full input packets and return empty packets; the same function must be able to supply both main image packets and overlay packets.

Data streaming is initiated using **tmalVrendVOStart**. The instance will call the datainFunc callback to obtain the main image input packet and possibly an overlay packet if the overlay is enabled.

The video renderer will then start rendering to the screen. Full packets are obtained in the video interrupt service routine using the datainFunc callback. As the video renderer is interrupt driven, the application is free to perform other operations. It may change the renderer parameters by calling **tmalVrendVOInstanceConfig** during this time; for example, to change the overlay position.

Data streaming can be terminated by calling **tmalVrendVOStop**. This will cause the instance to return any packets which it currently is using. The renderer will call the completionFunc callback to indicate that it has stopped. Note that the completion function is only called when it has stopped; this is different from the non-streaming case which calls the completion function after each frame has been displayed. Finally, the **tmalVrendVOClose** function closes the instance.

Note that it is possible to start VrendVO, calling **tmalVrendVOStart** without having a video format already installed. The renderer will actually start once **tmalVrendVOReceiverFormat** is called by the application or by another library. This can be done by calling **tsaDefaultInstallFormat**, or through the progress function, with **tsaProgressFlagChangeFormat** set.

The OL Layer

The operating system layer only supports streaming operation. A diagram of the typical flow of control is shown in Figure 18.

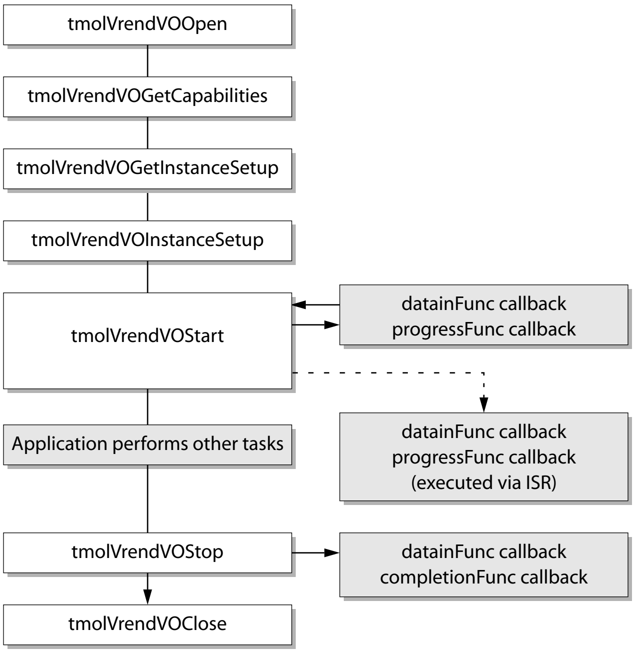


Figure 18 OL Layer Data Streaming Flow Control.

An instance of the video renderer should be opened first using **tmolVrendVOOpen**; only one instance is currently supported. The capabilities of the component should be obtained using **tmolVrendVOGetCapabilities**. This information will be used by the format manager to ensure that the two instances being connected together are compatible. The InOutDescriptor which connects the two components should then be created by initializing an **ptsaInOutDescriptorSetup_t** structure and calling **tsaDefaultInOutDescriptorCre-**

ate. This can also be used to automatically create packets which will be used to transfer data between component instances.

The pointer to the video renderer instance setup should be obtained using **tmolVrendVO-GetInstanceSetup**. This structure should be initialized with any application specific values, for example, the type of display adaptor being used. The application should then call **tmolVrendVOInstanceSetup** to configure the instance. Data streaming mode can then be initiated by calling **tmolVrendVOStart**. Image packets to be displayed are obtained using the datain call back function in exactly the same manner as described in the AL layer data streaming section.

The application can terminate data streaming using **tmolVrendVOStop**, and release the instance using **tmolVrendVOClose**. After the instance has been closed, the application should destroy the InOutDescriptor using the **tsaDefaultInOutDescriptorDestroy** function. This will automatically free the packets contained in the queues.

Callback Function Requirements

The following list gives the mandatory and optional callback functions used by the video renderer.

`datainFunc`	Used for data streaming in both the AL and OL layers. For AL Layer streaming the application must provide this function. For OL Layer streaming, the tsaDefaults library provides a default function automatically. This field must be set to Null for AL-Layer non-data streaming operation.
`completionFunc`	In AL Layer non-data streaming, this is called to indicate that a frame has been displayed. In AL Layer data-streaming and the OL Layer, this is used to indicate that streaming has stopped.
`errorFunc`	This is used in all layers to indicate that an error has occurred.
`progressFunc`	This is used to report progress: entering an ISR routine, reaching Ythreshold, losing a frame, reporting which Field is being processed in the ISR.

In streaming mode, datainFunc is mandatory, completionFunc, errorFunc and progressFunc are optional.

In non streaming mode, datainFunc has to be Null, completionFunc is mandatory, so that the video renderer can notify the application that a complete frame has been displayed. ErrorFunc and progressFunc are still optional.

Packet Formats

The video renderer uses the standard packet data types defined in the **tmAvFormats.h** include file. Both the main and overlay image use the **tmAvPacket_t** structure. The main image 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. The overlay image YUV sequence data will be stored in a single buffer.

Each packet contains a header structure providing information concerning the packet data. The format field will be a **tmVideoFormat_t** structure which specifies the format and the image size. There are restrictions on the type of video formats that can be accepted by the video renderer. These will be described next.

The description section of the **tmVideoFormat_t** structure enables the application to specify that the video stream is possibly interlaced, and that the video data are sent to the Video Renderer on a frame or field basis. It is also possible, using the description section to specify if the video packet has Mpeg extension. In that case, VrendVO will extract Mpeg related display information from the header->userPointer field of each incoming packet. This information is used by VrendVO to perform 3:2 pulldown. In this case also, VrendVO will automatically center and scale the image if necessary.

Main Image Input Packet

The main image input packet must be either YUV422 or YUV420. The packet headers format field should be initialized with the following values:

`dataClass`	avdcVideo.
`dataType`	vtfYUV.
`dataSubtype`	vdfYUV422Planer, vdfYUV420Planer, or vdfYUV422Interspersed
`description`	vdfInterlaced, vdfFieldInFrame, vdfFieldInField, vdfProgressive, or vdfMpegExtension.
`imageWidth`	Width of video frame (luminance).
`imageHeight`	Height of video frame (luminance).
`imageStride`	Stride of video frame (luminance).
`activeVideoStartX`	0
`activeVideoStartY`	0
`activeVideoEndX`	Width of video frame (luminance).
`activeVideoEndY`	Height of video frame (luminance).
`videoStandard`	vasNTSC or vasPAL.

Overlay Image Input Packet

The overlay image must be YUV sequence data. The format field should be initialized with the following values:

`dataClass`	avdcVideo.
`dataType`	vtfYUV.
`dataSubtype`	vdfYUVSequence or vdfYUVSequenceAlpha.

`Description`	vdfInterlaced, vdfFieldInField, vdfFieldInFrame, or vdfProgressive.
`imageWidth`	Width of overlay frame.
`imageHeight`	Height of overlay frame.
`imageStride`	Stride of overlay frame.
`activeVideoStartX`	0
`activeVideoStartY`	0
`activeVideoEndX`	Width of overlay frame.
`activeVideoEndY`	Height of overlay frame.
`videoStandard`	vasNTSC or vasPAL.

Cache Coherency

When the application is using the video renderer AL Layer, it must consider cache coherency issues. For example, if the DSPCPU created or manipulated the video data, then the application must perform a cache copyback operation on the data before passing it to the renderer. This can simply be performed using the **_cache_copyback** function on the relevant video data. Cache aligned video buffers can easily be created and destroyed using the **_cache_malloc** and **_cache_free** functions respectively.

When using the OL Layer of the video renderer, all cache coherency issues are automatically handled by the tsaDefaults library.

Video Renderer API Data Structures

This section describes the Video Renderer application library data structures. These data structures are defined in the tmalVrendVO.h and tmolVrendVO.h header files.

Name	Page
tmalVrendVOProgressFlags_t	188
tmalVrendVOCapabilities_t	189
tmolVrendVOCapabilities_t	189
tmalVrendVOInstanceSetup_t	190
tmolVrendVOInstanceSetup_t	190
tmalVrendVOConfigTypes_t	192

tmalVrendVOProgressFlags_t

```
typedef enum {
   VRENDVO_PROGRESS_YTHRESHOLD   = 0x00000001,
   VRENDVO_PROGRESS_REPORT       = 0x00000002,
   VRENDVO_PROGRESS_ISR_ENTRY    = 0x00000004,
   VRENDVO_PROGRESS_LOST_FRAME   = 0x00000008,
   VRENDVO_PROGRESS_FIELD1       = 0x00000010,
   VRENDVO_PROGRESS_FIELD2       = 0x00000020,
   VRENDVO_PROGRESS_TIMEDIFF     = 0x00000040,
} tmalVrendVOProgressFlags_t;
```

Description

This enumerated type describes the flags that are used when the progress function is called by the renderer. Those flags can be OR'd, so that the application can choose which progress needs to be reported by the video render component.

tmalVrendVOCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
   Int                         granularityOfAddress;
   Int                         granularityOfStride;
   UInt32                      videoStandards;
   UInt32                      adapterTypes;
} tmalVrendVOCapabilities_t, *ptmalVrendVOCapabilities_t;
```

tmolVrendVOCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
   Int                         granularityOfAddress;
   Int                         granularityOfStride;
   UInt32                      videoStandards;
   UInt32                      adapterTypes;
} tmlVrendVOCapabilities_t, *ptmolVrendVOCapabilities_t;
```

Fields

`defaultCapabilities`	Pointer to the default capabilities structure.
`granularityOfAddress`	Number of bits which must be zero in the address of YUV data.
`granularityOfStride`	Number of bits which must be zero in the stride.
`videoStandards`	OR'd values of different video standards supported by the VO device library.
`adapterTypes`	OR'd values of different adapter types supported by the VO device library.

Description

This structure is used to specify the capabilities of the video renderer. An application can obtain the components capability structure by calling **tmalVrendVOGetCapabilities** at the AL layer, or **tmolVrendVOGetCapabilities** at the OL layer.

tmalVrendVOInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   tmVideoAnalogStandard_t     videoStandard;
   tmVideoAnalogAdapter_t      adapterType;
   Bool                        scaleUp;
   Bool                        overlayEnable;
   UInt16                      imageHorzOffset;
   UInt16                      imageVertOffset;
   UInt16                      overlayHorzOffset;
   UInt16                      overlayVertOffset;
   UInt16                      overlayAlpha0;
   UInt16                      overlayAlpha1;
   Bool                        hbeEnable;
   Bool                        underrunEnable;
   Bool                        yThresholdEnable;
   UInt32                      yThreshold;
   Bool                        underrunHoldFields;
} tmalVrendVOInstanceSetup_t, *ptmalVrendVOInstanceSetup_t;
```

tmolVrendVOInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   tmVideoAnalogStandard_t     videoStandard;
   tmVideoAnalogAdapter_t      adapterType;
   Bool                        scaleUp;
   Bool                        overlayEnable;
   UInt16                      imageHorzOffset;
   UInt16                      imageVertOffset;
   UInt16                      overlayHorzOffset;
   UInt16                      overlayVertOffset;
   UInt16                      overlayAlpha0;
   UInt16                      overlayAlpha1;
   Bool                        hbeEnable;
   Bool                        underrunEnable;
   Bool                        yThresholdEnable;
   UInt32                      yThreshold;
   Bool                        underrunHoldFields;
} tmolVrendVOInstanceSetup_t, *ptmolVrendVOInstanceSetup_t;
```

Fields

`defaultSetup`	Default instance setup (see tsa.h). Note that the AL and OL layers are identical.
`videoStandard`	vasPAL or vasNTSC.
`adapterType`	CVBS or S-Video. see tmAvFormats.h

`scaleUp`	True if this is an MPEG-1 type SIF image which should be doubled in size by the video out hardware.
`overlayEnable`	True if the overlay functionality will be used.
`imageHorzOffset`	Specified in pixels from the left edge. Note that an offset of zero is likely to be displayed off screen on most video monitors.
`imageVertOffset`	Specified in lines from the top of the screen.
`overlayHorzOffset`	Specified in pixels from the left edge.
`overlayVertOffset`	Specified in pixels from the top of the screen.
`overlayAlpha0`	Specifies the alpha value to use when the alpha bit is zero.
`overlayAlpha1`	Specifies the alpha value to use when the alpha bit is one.
`hbeEnable`	Set to True to turn on highway bandwidth interrupts.
`underrunEnable`	Set to True to turn on underrun interrupts.
`yThresholdEnable`	Set to True to turn on yThreshold interrupts.
`yThreshold`	If yThreshold interrupts are turned on, this field contains the value of the line in the video buffer that will trigger the interrupt.
`underrunHoldFields`	Used in field mode when a packet underrun occurs. If false, the most recent field is re-displayed regardless of whether it is the correct field type. If true, the instance keeps both top and bottom field packets so the correct field type is displayed.

Description

This structure can be used by the application to set up the initial configuration of the video renderer. In the AL Layer, the application should create and initialize the structure and then call **tmalVrendVOInstanceSetup**. In the OL Layer, the application should call **tmolVrendVOGetInstanceSetup** to obtain a pointer to the structure. It may then initialize any specific values before calling **tmolVrendVOInstanceSetup**.

tmalVrendVOConfigTypes_t

```
typedef enum {
   VO_CONFIG_SET_DDS_FREQUENCY          = tsaCmdUserBase + 0x01,
   VO_CONFIG_SET_OVERLAY                = tsaCmdUserBase + 0x02,
   VO_CONFIG_SET_OVERLAY_HORZ_OFFSET    = tsaCmdUserBase + 0x03,
   VO_CONFIG_SET_OVERLAY_VERT_OFFSET    = tsaCmdUserBase + 0x04,
   VO_CONFIG_SET_HORZ_OFFSET            = tsaCmdUserBase + 0x05,
   VO_CONFIG_SET_VERT_OFFSET            = tsaCmdUserBase + 0x06,
   VO_CONFIG_SET_YTHRESHOLD             = tsaCmdUserBase + 0x07,
   VO_CONFIG_DES_YTHRESHOLD             = tsaCmdUserBase + 0x08,
   VO_CONFIG_SET_MPEG_PLAY              = tsaCmdUserBase + 0x09,
   VO_CONFIG_SET_MPEG_PAUSE             = tsaCmdUserBase + 0x0a,
   VO_CONFIG_SET_MPEG_SFA               = tsaCmdUserBase + 0x0b,
   VO_CONFIG_SET_MPEG_IGNORE_PTS        = tsaCmdUserBase + 0x0c,
   VO_CONFIG_SET_WINDOW                 = tsaCmdUserBase + 0x0d,
   VO_CONFIG_GET_DDS_FREQUENCY          = tsaCmdUserBase + 0x81,
   VO_CONFIG_GET_OVERLAY                = tsaCmdUserBase + 0x82,
   VO_CONFIG_GET_OVERLAY_HORZ_OFFSET    = tsaCmdUserBase + 0x83,
   VO_CONFIG_GET_OVERLAY_VERT_OFFSET    = tsaCmdUserBase + 0x84,
   VO_CONFIG_GET_HORZ_OFFSET            = tsaCmdUserBase + 0x85,
   VO_CONFIG_GET_VERT_OFFSET            = tsaCmdUserBase + 0x86,
   VO_CONFIG_GET_YTHRESHOLD             = tsaCmdUserBase + 0x87,
   VO_CONFIG_GET_MPEG_PTS               = tsaCmdUserBase + 0x88
} tmalVrendVOConfigTypes_t, *ptmalVrendVOConfigTypes_t;
```

Commands

`VO_CONFIG_SET_DDS_FREQUENCY`	Change frequency.
`VO_CONFIG_SET_OVERLAY`	Allow user to set overlay on and off.
`VO_CONFIG_SET_OVERLAY_HORZ_OFFSET`	Change horizontal position of overlay.
`VO_CONFIG_SET_OVERLAY_VERT_OFFSET`	Change vertical position of overlay.
`VO_CONFIG_SET_HORZ_OFFSET`	Change horizontal offset of main image. Offset is specified in pixels from the left edge.
`VO_CONFIG_SET_VERT_OFFSET`	Change vertical offset of main image. Offset is specified in lines from the top of the screen.
`VO_CONFIG_SET_YTHRESHOLD`	Enables **yThreshold** interrupts and sets a new value for **yThreshold**.
`VO_CONFIG_DES_YTHRESHOLD`	Disables **yThreshold** interrupts.
`VO_CONFIG_SET_MPEG_PLAY`	Acquire and display video packets.
`VO_CONFIG_SET_MPEG_PAUSE`	Repeatedly display the same frame without acquiring new packets.
`VO_CONFIG_SET_MPEG_SFA`	Single-frame advance: obtain a new video packet and repeatedly display it.

`VO_CONFIG_SET_MPEG_IGNORE_PTS`	Ignore any PTS timestamps and display the video frame immediately.
`VO_CONFIG_SET_WINDOW`	Reserved for future use.

Description

This enumeration type describes the *command* field of the **tsaControlArgs_t** structure, that is used as a parameter by the **tmalVrendVOInstanceConfig** and **tmolVrendVOInstanceConfig** functions to change certain instance parameters while the renderer is running.

Video Renderer API Functions

This section describes the TriMedia Video Renderer application library API functions.

Name	Page
tmalVrendVOGetCapabilities	195
tmolVrendVOGetCapabilities	195
tmalVrendVOOpen	196
tmolVrendVOOpen	196
tmalVrendVOClose	197
tmolVrendVOClose	197
tmolVrendVOGetInstanceSetup	198
tmalVrendVOInstanceSetup	199
tmolVrendVOInstanceSetup	199
tmalVrendVOStart	200
tmolVrendVOStart	200
tmalVrendVOStop	201
tmolVrendVOStop	201
tmalVrendVOInstanceConfig	202
tmolVrendVOInstanceConfig	202
tmalVrendVORenderFrame	203
tmalVrendVORenderOverlay	204
tmalVrendVOReceiverFormat	205

tmalVrendVOGetCapabilities

```
tmLibappErr_t tmalVrendVOGetCapabilities(
   tmalVrendVOCapabilities_t   **cap
);
```

tmolVrendVOGetCapabilities

```
tmLibappErr_t tmolVrendVOGetCapabilities(
   ptmolVrendVOCapabilities_t   *pCap
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.

Description

These functions fill in the pointer of a static **tmalVrendVOCapabilities_t** structure maintained by the renderer to describe the capabilities and requirements of this library.

tmalVrendVOOpen

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

tmolVrendVOOpen

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

Parameters

`instance`	Pointer to the (returned) instance.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MODULE_IN_USE`	Maximum number of renderers are allocated.

Description

Instantiates a video renderer, and sets the instance variable to point to the video renderer instance. Currently only one instance is supported.

tmalVrendVOClose

```
tmLibappErr_t tmalVrendVOClose(
   Int   instance
);
```

tmolVrendVOClose

```
tmLibappErr_t tmolVrendVOClose(
   Int   instance
);
```

Parameters

`instance`Instance value, as returned by **tmalVrendVOOpen** or **tmolVrendVOOpen**.

Return Codes

`TMLIBAPP_OK`	Success
`TMLIBAPP_ERR_MODULE_IN_USE`	Asserts if the renderer has not been opened by this instance.

Description

These functions will shut down an instance of the renderer. The instance must have been stopped prior to calling the respective function.

tmolVrendVOGetInstanceSetup

```
tmLibappErr_t tmolVrendVOGetInstanceSetup(
   Int                           instance,
   tmolVrendVOInstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance value, as returned by **tmolVrendVOOpen**.
`setup`	Pointer to the setup structure (see page 190).

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MODULE_IN_USE`	If the renderer has not been opened by this instance.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Can assert if the desired instance is not open.
`VR_ERR_DEVICE_LIBRARY_ERROR`	Is OR'd with the low byte of the return code of the device library if **voInstanceSetup** fails.

Description

The **tmolVrendVOGetInstanceSetup** function is used to return a pointer to the renderer's OL Layer instance setup structure. The renderer creates this structure when the component is opened. After obtaining the pointer to the structure, the application can initialize specific instance values before calling **tmolVrendVOInstanceSetup**.

Default values for the returned instance setup are shown below:

```
defaultSetup       == defaultSetup
videoStandard      == vasNSTC
adapterType        == vaaCVBS
scaleUp            == False
overlayEnable      == False
imageHorzOffset    == 0
imageVertOffset    == 0
overlayHorzOffset  == 0
overlayVertOffset  == 0
overlayAlpha0      == 0
overlayAlpha1      == 0
hbeEnableTrue,     == True
underrunEnable     == True
yThresholdEnable   == False
yThreshold         == 0
underrunHoldFields == False
```

tmalVrendVOInstanceSetup

```
tmLibappErr_t tmalVrendVOInstanceSetup(
   Int                          instance,
   tmalVrendVOInstanceSetup_t   *setup
);
```

tmolVrendVOInstanceSetup

```
tmLibappErr_t tmolVrendVOInstanceSetup(
   Int                          instance,
   tmolVrendVOInstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance value, as returned by **tmalVrendVOOpen** or **tmolVrendVOOpen**.
`setup`	Pointer to the setup structure (see page 190).

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MODULE_IN_USE`	If the renderer has not been opened by this instance.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Can assert if the desired instance is not open.
`VR_ERR_DEVICE_LIBRARY_ERROR`	Is OR'd with the low byte of the return code of the device library if **voInstanceSetup** fails

Description

These functions configure the renderer. The video-out device will be opened, and the renderer will be in a stopped state. After initialization, the application should use the **tmalVrendVOInstanceConfig** and **tmolVrendVOInstanceConfig** functions to modify instance variables.

tmalVrendVOStart

```
tmLibappErr_t tmalVrendVOStart(
   Int   instance
);
```

tmolVrendVOStart

```
tmLibappErr_t tmolVrendVOStart(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmalVrendVOOpen** or **tmolVrendVOOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the desired instance is not open.
`VR_ERR_DEVICE_LIBRARY_ERROR`	OR'd with the low byte of the return code of the device library if voInstanceSetup fails
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the instance variable is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	Asserts if the instance variable does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the instance has not been configured using the instance setup functions.

Description

These functions start the video rendering for the specific instance. In OL Layer or AL Layer streaming mode, the datain function is called to obtain the initial main image packet, and optionally the overlay packet. The VO module is then started.

In AL Layer non-streaming mode, the function simply returns.

tmalVrendVOStop

```
tmLibappErr_t tmalVrendVOStop(
   Int   instance
);
```

tmolVrendVOStop

```
tmLibappErr_t tmolVrendVOStop(
   Int   instance
);
```

Parameters

`instance`	Instance, as returned by **tmalVrendVOOpen** or **tmolVrendVOOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MODULE_IN_USE`	Asserts if the desired instance is not open.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the instance variable is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	Asserts if the instance variable doe not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the instance has not been configured using the instance setup functions.

Description

These functions stop the video renderer and call **voStop**.

In AL Layer non-streaming mode, any packets held in the internal queue are returned; the completion function is called for each packet on the queue with the completion function flags being set to the packet ID.

In AL Layer streaming and the OL Layer, the renderer can only hold a single main image packet and one overlay packet (if the overlay is enabled). The function will return the respective packet using the datainFunc callback, and call the completion function for each returned packet.

tmalVrendVOInstanceConfig

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

tmolVrendVOInstanceConfig

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

Parameters

`instance`	Instance value, as returned by **tmalVrendVOOpen** or **tmolVrendVOOpen**.
`args`	Pointer to **tsaControlArgs_t** structure. Two fields of this structure are used to update the instance configuration: *command* and *parameter.*

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_MODULE_IN_USE`	Asserts if the desired instance is not open.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the instance variable is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	Asserts if the instance variable does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the instance has not been configured using the instance setup functions.
`VRENDVO_ERR_CONFIG_UNKNOW_COMMAND`	**tmalVrendVOInstanceConfig** has been called with an invalid command.

Description

These functions can be used to change instance parameters after the component has been initialized and during streaming operation. For example, the overlay enable flag can be changed, or the overlay position moved.

The control structures *command* field should be set to one of the command values specified by the enumeration **tmalVrendVOConfigTypes_t** on page 192. When a parameter is required, its value should be passed in the control structures *parameter* field.

tmalVrendVORenderFrame

```
tmLibappErr_t tmalVrendVORenderFrame(
   Int              instance,
   tmYuvPacket_t   *frame
);
```

Parameters

`instance`	Instance, from **tmalVrendVOOpen**.
`frame`	Pointer to a packet of video data.

Return Codes

`TMLIBAPP_OK`	Success.
`VR_ERR_PUSH_PULL_CONFUSION`	A **datainFunc** is installed. Render frame is not used in streaming mode.
`VR_ERR_NO_MORE_NODES`	The instance already has four packets to be rendered on its internal queue.
`VR_ERR_DEVICE_LIBRARY_ERROR`	OR'd with the low byte of the return code of the device library if **voInstanceSetup** fails.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Asserts if the instance variable passed is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	Asserts if the instance variable passed does not match the currently opened instance.
`TMLIBAPP_ERR_NOT_SETUP`	Asserts if the instance has not been configured using the instance setup functions.
`VR_ERR_INVALID_ADDRESS`	Asserts if the video buffer is not 64-byte aligned.

Description

In non-streaming mode, this function is used to pass a frame from the application to the renderer for display. The frame will be displayed using the settings assigned using the **tmalVrendVOInstanceSetup** and **tmalVrendVOInstanceConfig** functions.

The completion callback function will be called when this frame has been displayed and there are more frames to be displayed on the internal queue. In non-streaming mode, the video renderer keeps an internal queue of up to four packets. Invocation of this function when the queue is full will have no effect and will return **VR_ERR_NO_MORE_NODES**.

tmalVrendVORenderOverlay

```
tmLibappErr_t tmalVrendVORenderOverlay(
   Int             instance,
   tmAvPacket_t   *frame
);
```

Parameters

`instance`	Instance value, from **tmalVrendVOOpen**.
`frame`	Pointer to a packet of video data

Return Codes

`TMLIBAPP_OK`	Success.
`VR_ERR_PUSH_PULL_CONFUSION`	If a **datainFunc** is installed. Render frame is not used in streaming mode.
`VR_ERR_NO_MORE_NODES`	If instance is not yet ready for new data
`VR_ERR_DEVICE_LIBRARY_ERROR`	Is OR'd with the low byte of the return code of the device library if voInstanceSetup fails.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Can assert if the instance variable is Null.
`TMLIBAPP_ERR_MODULE_IN_USE`	Can assert if the instance variable does not match the currently opened instance
`TMLIBAPP_ERR_NOT_SETUP`	Can assert if the instance has not been configured using the instance setup functions.
`VR_ERR_INVALID_ADDRESS`	Can assert if the overlay buffer is not 64-byte aligned.

Description

This function is used to change the video image assigned to the overlay surface. Note that the overlay must be in sequential YUV422 format.

tmalVrendVOReceiverFormat

```
tmLibappErr_t tmalVrendVOReceiverFormat(
   UInt32              inputIndex,
   ptmVideoFormat_t    format
)
```

Parameters

`inputIndex`	If **inputIndex** = **VRENDVO_MAIN_INPUT**, the format must be installed on the main input of the Video Renderer. If **inputIndex** = **VRENDVO_OVERLAY_INPUT**, the format must be installed on the overlay input of the Video Renderer.
`format`	Pointer to the video format that needs to be installed on the queue.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_FORMAT_NULL_FORMAT`	Asserts if format is null.
`VR_ERR_IMAGE_FORMAT`	Asserts if **format–>datasubtype** is not supported by VrendVO.
`VR_ERR_IMAGE_WIDTH`	Asserts if (**format->activeVideoEndX** – **format–>activeVideoStartX**) > **format–>imageWidth**, or if **format–>imageWidth** = **0**.
`VR_ERR_IMAGE_HEIGHT`	Asserts if (**format–>activeVideoEndY** – **format–>activeVideoStartY**) > **format–>imageHeight**, or if **format->imageHeight** == **0**.

Description

This function is used by the application or a sender component to install a new format for the Video Renderer. A sender component can call the **tsaDefaultProgressFunction** with the flag set to **tsaProgressFlagChangeFormat**. This calls **tmalVrendVOReceiverFormat** and install the new format on the queue. An application can also call the **tsaDefault-InstallFormat** function, which also calls **tmalVrendVOReceiverFormat**.

In case the video standard has changed, **tmalVrendVOReceiverFormat** stops the Video Render, does an instance setup, installs the new standard and restarts the Video Renderer automatically. Any other changes are made without stopping the Video Renderer.