Book 5—System Utilities

Part C:

System Device Libraries

TriMedia

Version 2.0 beta

Table of Contents

Chapter 19 TMBoard API

Chapter 20 Exceptions API

Chapter 21 TriMedia Interrupts API

Chapter 25 Timers API

Chapter 26 DMA API

Chapter 19

TMBoard API

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

(The Board Support interface was enhanced significantly for TCS v2.0 beta.)

Board Support API Overview

The TriMedia Board Support API is used to install handlers, or drivers for the parts of the TriMedia software that rely on hardware off of the chip. Since this hardware is not on the chip, it must be "on the board" instead. Things that fall into this category include system reset, audio, video and telecom hardware. The software layer that describes the interface between the software on chip and any peripherals on the board is known as the Board Support Package, or BSP.

A complete BSP uses the services of the TriMedia Registry and the TriMedia Component Manager (see Part A, Chapter 2 and Chapter 3 respectively), and it is not limited to the functions in this library. But for historical reasons, the components most closely related to on-chip TriMedia peripherals are managed though this board library.

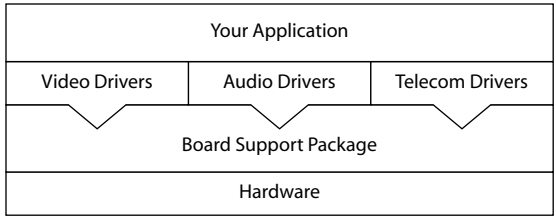

Figure 1 BSP Hides Hardware Details

Note
The device libraries and board support packages form a software layer that is below the traditional device driver. An engineer could use the BSP to construct a device driver in the tradition of UNIX and pSOS. In the TSSA model, the device drivers that reside above the device library are the renderers and digitizers.

Why BSP?

The BSP allows you to change the board's design without affecting the software that has already been developed. If you decide to change the board's design, you would only have to change the implementation of the functions described in this chapter.

For example, the IREF board uses the *Analog Devices AD1847* as an audio codec. If you want to use a different device, the code that you would need to change is the BSP. This mechanism is implemented through a table of function pointers that can be retrieved from the registry. This table is used by the device libraries, and not directly by applications.

Components of a Board Support Package

A BSP has three components:

- Core board library
- Board initialization function
- Component export macro

The Core Board Library

Through a number of functions, the core board library provides upper layers of software with information about the existing board. To an application, the most important of these is as shown below:

```
tsaBoardGetBoard(UInt32 *pID, Char **pboardName);
```

This function can be called by libraries or applications that want to know on what board they are running. On success, the board ID will be as specified in the EEPROM used to boot the TriMedia. At system bootup, the board ID is read from the EEPROM and software is chosen to support the identified board. When you call **tsaBoardGetBoard**, the board ID and name (as determined at bootup) are returned.

In addition, a collection of functions is used to support existing TriMedia hardware peripherals that require off-chip support. These functions come in register/get pairs. For each peripheral, the *register* function is to be called as part of the board's initialization sequence. The *get* function is used by the corresponding TriMedia device library. For backward compatibility, the obsolete function **boardGetID** is retained. New code should use `tsaBoardGetBoard` instead.

The core of the board library can be considered to also include TriMedia's Registry and Component Manager. Users do not modify the core of the board library.

The Board Initialization Function

Each supported board will export an initialization function. This code will be called by the component manager before program startup. The initialization function uses the registry to inform the system of the capabilities of a given board. An example of a board initialization function is described below.

The Board's Component Export Macro

For the component manager to find the support package for your board, you must include the macro **TSA_COMP_DEF_O_COMPONENT** with appropriate parameters. This macro causes symbols to be exported so that the component manager can find your board as a component.

How a Board Support Package is Initialized

The BSP identifies itself to the component manager using the macro referenced above. At boot time, and before **main** is called, the component manager initializes each component. Boards include a detect function, and this is called to decide whether the current BSP matches the board. The decision is based upon the board ID given in the EEPROM.

If the board ID matches, this BSP is accepted, and the audio, video and telecom libraries are free to use the information provided by the BSP.

How a Board Support Package is Delivered

The standard device library is delivered in the archive libdev.a. The archive contains a number of files that are BSP specific:

- tmBoard.o—The core board library functions.
- 6ch.o, ad1847.o, tda1315.o—Routines used to support common audio I/O.
- saa7111.o, saa7112.o, saa7113.o, saa7125.o, saa7182.o, saa7185.o, st7545.o, voSupport.o—routines that support common video I/O.

These routines are referred to as the *common* portion of the BSP. They are routines used by more than one board.

Also in the library directory are object files supporting a specific set of boards:

- libBSPiref.o—the traditional reference board.
- libBSPdtv_ref2.o, libBSPdtv_ref3.o, libBSPdtv_ref4.o, libBSPnim.o—reference boards used by Philips DTV.
- libBSPdebug_tm1000.o, libBSPdebug_tm1100.o—boards used within Philips to debug hardware.

Of these, the IREF and the DTV_REF2 board are linked with user programs by default, using the tmconfig file. Support for other boards can be added to an application simply by adding the required object file to the program's link line. Object files are used instead of archive files because the chaining mechanism used by the component manager requires it. Each BSP should be provided as a single object file. No header file is required.

How To Support a New Board

Any time that new hardware functionality is provided, a system designer must also create a board support package to serve as the lowest level of interface to the board. This chapter will describe the BSP for the DTV reference board, as an example. The DTV board is used instead of the IREF because it is slightly more complex, and as a result, illustrates more concepts.

Files That Make Up the BSP

The support for the DTV REF2 board resides in two C files. Any number of source files could be used, including one. It is advantageous to use only one C file, as all functions can be declared static, thereby reducing name space pollution. A BSP is not required to export any symbols, except for a hidden symbol that is exported by the component manager macro. In the past, each board support package was required to provide an header file describing its exports. When using the component manager, this is no longer the case. Also of interest is the makefile, in particular because of the way that it can combine several object files into a single object file. To do this, the linker is invoked using a rule as follows:

```
libBSPdtv_ref2.o : dtv_ref2_audio.o philips_dtv_ref2.o
   tmld dtv_ref2_audio.o philips_dtv_ref2.o -o libBSPdtv_ref2.o
```

Note that a BSP can also be delivered as a DLL, and that all of the default BSP's are provided in both debuggable (end with _g), and stripped versions.

In the DTV BSP, the functionality is split between audio and everything else.

Minimum Requirement for a BSP

A BSP must export an output using the component manager macro, as in this example:

```
TSA_COMP_DEF_O_COMPONENT(
   Philips_dtv_ref2,
   TSA_COMP_BUILD_ARG_LIST_1("bsp/boardID"),
   dtv_ref2_board_activate );
```

The first argument is the name of the board, and it is treated as a string. This name is returned by the **tsaBoardGetBoard** function, and it is printed by the function in the **tmHelp** package. The second argument tells the component manager that the output symbol is to be called "bsp/boardID." The third argument is the name of the activation function that will be called to initialize this component.

Hence, the minimum requirement is the export of the bsp/boardID symbol, and the activation function.

Role of the Activation Function

The activation function must first check the board to determine whether this is the correct activation function for this board. Usually, this is done by reading the EEPROM. If the board ID does not match what is expected (as defined in tmBoardID.h), the activation function returns False, and the component manager can try another component.

The activation function will then run any code necessary to bring the board out of reset. On many TriMedia boards, this includes writing to an IIC location to raise a hardware reset pin. Finally, the activation function registers the various features of the board. At the least, this includes things like the audio and video, but it also should include things like UARTs, or hardware MPEG decoders that are on the board and that require software support.

Assignment of boardID

The board ID is a unique 8-bit number that is used to identify your board to supporting software. It is stored in the boot EEPROM, at locations 1 and 2. The term *relatively* is used because a number of things influence the choice of a board ID. Systems are actually identified by the combination of the board ID and the vendor ID, which is stored in locations 3 and 4. If the vendor is not Philips Semiconductors, then the 0x1131 vendor ID is incorrect. Change that to reflect the correct vendor, and then the assignment of board IDs is up to you.

If your board is being built by Philips, then you can ask whether this board will be publicly available in open PCI systems. If not, you may as well just choose a rather arbitrary ID, as your group of engineers are the only people who will see it. Choose an ID from the range above **BOARD_VERSION_RESERVED_FOR_PRIVATE_USE_BASE**.

If your board is being built by Philips and it will be publicly available, it is probably appropriate to contact the TriMedia developers and have us assign a range of IDs.

Creating Support for Audio and Video

Device library components such as audio and video that make use of the board support package call the **tsaBoardRegister*XX*** functions to tell the system about the functions that support a given component. In general, there are functions for initialization, termination, and control. More information is given in the API descriptions beginning on page 17. More information about the audio BSP can be found in Chapter 2 of Book 6. The SDE disk includes the source for a number of board support packages. These are provided so you can use them as examples.

Creating Support for Non-Standard Board Components

An examination of the **tsaBoardRegister*XX*** functions will show you that all they do is add keys to the registry. Using bsp/yourComponent as the path, you can follow this example and add your own keys to the registry. Then, just make sure that your application code uses the registry to retrieve the board support functions.

TMBoard API Data Structures

This section describes the TMBoard API data structures. Component-specific structures are defined in component-specific header files. For example, **boardAIParam_t** is defined in tmAIboard.h.

A. This structure should not be used in new code and is being maintained for compatibility purposes only. Instead, use **tsaBoardGet*** functions to retrieve this information.

boardAIParam_t

```
typedef struct {
   tmAudioTypeFormat_t     audioTypeFormat;
   UInt32                  audioSubtypeFormat;
   UInt32                  audioDescription;
   Float                   sRate;
   Int                     size;
   tmAudioAnalogAdapter_t  input;
} boardAIParam_t;
```

Fields

`audioTypeFormat`	The audio type as defined in tmAvFormats.h, usually **atfLinearPCM**.
`audioSubtypeFormat`	The audio subtype formal as defined in tmAvFormats.h, for instance a common one is **apfStereo16**.
`audioDescription`	Additional description fo the audio data.
`sRate`	Sample rate [Hertz].
`size`	Used to set the size register in the audio output section.
`input`	Input of the type **tmAudioAnalogAdapter_t**. (See Chapter 4 of Book 3, *Software Architecture*, Part A.)

Description

Used by the tmAI device library to initialize the board component of the audio-in system. It allows you to choose between multiple inputs (or outputs) on the same board.

boardAIConfig_t

```
typedef struct{
   Char              codecName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t   (*initFunc  )( pboardAIParam_t param );
   tmLibdevErr_t   (*termFunc  )( void );
   tmLibdevErr_t   (*startFunc )( void );
   tmLibdevErr_t   (*stopFunc  )( void );
   tmLibdevErr_t   (*setSRate  )( Float sRate );
   tmLibdevErr_t   (*getSRate  )( Float *sRate );
   tmLibdevErr_t   (*setVolume )( Int lGain, Int rGain );
   tmLibdevErr_t   (*getVolume )( Int *lGain, Int *rGain );
   tmLibdevErr_t   (*setInput  )( tmAudioAnalogAdapter_t input );
   tmLibdevErr_t   (*getInput  )( tmAudioAnalogAdapter_t *input );
   tmLibdevErr_t   (*configFunc)( UInt32 subAddr, Pointer value );
   tmLibdevErr_t   (*getFormat )( tmAudioFormat_t *format );
   UInt32            audioTypeFormats;
   UInt32            audioSubtypeFormats;
   UInt32            audioAdapters;
   intInterrupt_t    intNumber;
   UInt32            mmioBase;
   Float             maxSRate;
   Float             minSRate;
   UInt32            gpioFirstPin;
   UInt32            gpioLastPin;
} boardAIConfig_t;
```

Fields

`codecName`	Codec name, in human-readable form.
`initfunc`	This is the function called in **aiInstanceSetup**. On success, it must leave the audio input system "stopped" but otherwise ready for action. Board- and/or codec-specific actions, such as setting IIC control bits or initializing codec registers, are performed. The function takes one input parameter, a structure used to initialize the codec. It has fields **audioTypeFormat** and **audioSubtypeFormat**. It sets the minimum and maximum sample rates supported by the codec.
`termfunc`	This is called in **aiClose**. It should leave the audio input system shut down.
`startFunc`	Called from **aiStart**.
`stopFunc`	Called from a**iStop**.
`setSRate`	Called from **aiSetSRate**. It takes one parameter **sRate** and sets the sample rate of the codec to that value.

`getSRate`	Called from **aiGetSRate**. It sets one parameter **sRate** to the current sample rate of the codec.
`setVolume`	Called from **aiSetVolume**. It takes two parameters, **lgain** and **rgain**, which are the new left and right speaker volume settings.
`setInput`	Called from **aiSetInput**. It takes one input parameter **input** which is the audio input source, such as **aaaMicInput**.
`configFunc`	Called from **aiConfigure**, and it is a "backdoor" to support features not foreseen in the API. It takes two parameters which could be the IIC address and value for codec specific settings.
`getFormat`	Used to report the format of the incoming audio. This is to be used with digital input (like S/PDIF) where the format of the incoming data is not known in advance.
`audioTypeFormats`	This is an OR'd bitmask of audio types (**audioTypeFormats_t**) supported by the codec. It is initialized by the board's initialization function and is reported by the **aiGetCapabilities** function.
`audioSubtypeFormats`	This is an OR'd bitmask of audio subtypes (**audioSubTypeFormats_t**) supported by the codec. It is initialized by the board's init function and is reported by the **aiGetCapabilities** function.
`audioAdapters`	Which audio input adapters are available (an OR'd combination of **tmAudioAnalogAdapter_t**).
`intNumber`	Because multiple audio inputs are supported, this field tells the software which interrupt to use.
`mmioBase`	Because multiple audio inputs are supported, this field tells the software which set of MMIO registers to use.
`maxSRate`	Maximum sample rate.
`minSRate`	Minimum sample rate.
`gpioFirstPin`, `gpioLastPin`	On chips that support GPIO functionality, these two fields identify the pins used by this device.

Description

A struct of this type describes the capabilities of the audio input subsystem to the tmAI device library.

boardAOParam_t

```
typedef struct {
   tmAudioTypeFormat_t     audioTypeFormat;
   UInt32                  audioSubtypeFormat;
   UInt32                  audioDescription;
   Float                   srate;
   Int                     size;
   tmAudioAnalogAdapter_t  output;
} boardAOParam_t;
```

Fields

`audioTypeFormat`	This is an OR'd bitmask of audio types (**audioTypeFormats_t**) supported by the codec. It is initialized by the board's init function and is reported by the **aiGetCapabilities** function.
`audioSubtypeFormat`	This is an OR'd bitmask of audio subtypes (**audioSubTypeFormats_t**) supported by the codec. It is initialized by the board's init function and is reported by the **aiGetCapabilities** function.
`audioDescription`	Additional description of the audio data.
`srate`	Sample rate [Hertz].
`size`	Number of samples in buffer.
`output`	The output.

Description

Structures of this type are used by the tmAO device library to initialize the board component of the audio out system.

boardAOConfig_t

```
typedef struct{
   Char              codecName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t   (*initFunc  )( pboardAOParam_t setup );
   tmLibdevErr_t   (*termFunc  )( void );
   tmLibdevErr_t   (*startfunc )( void );
   tmLibdevErr_t   (*stopfunc  )( void );
   tmLibdevErr_t   (*setSRate  )( Float sRate );
   tmLibdevErr_t   (*getSRate  )( Float *sRate );
   tmLibdevErr_t   (*setVolume )( Int lgain, Int rgain );
   tmLibdevErr_t   (*getVolume )( Int *lGain, Int *rGain );
   tmLibdevErr_t   (*setOutput )( tmAudioAnalogAdapter_t output );
   tmLibdevErr_t   (*getOutput )( tmAudioAnalogAdapter_t *output );
   tmLibdevErr_t   (*configFunc)( UInt32 subAddr, Pointer value );
   UInt32            audioTypeFormats;
   UInt32            audioSubtypeFormats;
   UInt32            audioAdapters;
   intInterrupt_t    intNumber;
   UInt32            mmioBase;
   Float             minSRate;
   UInt32            gpioFirstPin;
   UInt32            gpioLastPin;
} boardAOConfig_t;
```

Fields

`codecName`	Name of the codec in human-readable format.
`initfunc`	This is the function called in **aoInstanceSetup**. On success, it must leave the audio input system "stopped" but otherwise ready for action. Board and/or codec specific actions like setting IIC control bits or initializing codec registers are performed. The function takes one input parameter. This is a structure used to initialize the codec. It has fields **audioTypeFormat** and **audioSubtypeFormat**, and it sets the minimum and maximum sample rates supported by the codec.
`termFunc`	Called from **aiClose**. It should leave the audio input system shut down.
`startfunc`	Called from **aoStart**.
`stopfunc`	Called from **aoStart**.
`setSRate`	Called from **aiSetSRate**.It takes one parameter **sRate**, and sets the sample rate of the codec to that value.
`getSRate`	Called from **aoGetSRate**. It sets one parameter **sRate** to the current sample rate of the codec.

`setVolume`	Called from **aoSetVolume**. It takes two parameters, **lgain** and **rgain**, which are the new left and right speaker volume settings.
`getVolume`	Called from **aoGetVolume**.
`setOutput`	Called from **aoSetInput**. It takes one input parameter input which is the audio input source, such as **aaaMicInput**.
`getOutput`	Called from **aoGetInput**.
`configFunc`	Called from **aoConfigure**, and it is a backdoor to support features not foreseen in the API. It takes two parameters which could be the IIC address and value for codec specific settings.
`audioTypeFormats`	This is an OR'd bitmask of audio types (**audioTypeFormats_t**) supported by the codec. It is initialized by the board's init function and is reported by the **aoGetCapabilities** function.
`audioSubtypeFormats`	This is an OR'd bitmask of audio subtypes (**audioSubTypeFormats_t**) supported by the codec. It is initialized by the board's init function and is reported by the **aoGetCapabilities** function.
`audioAdapters`	Which audio input adapters are available (an OR'd combination of **tmAudioAnalogAdapter_t**).
`intNumber`	Because multiple audio inputs are supported, this field tells the software which interrupt to use.
`mmioBase`	Because multiple audio inputs are supported, this field tells the software which set of MMIO registers to use.
`minSRate`	Minimum sample rate.
`gpioFirstPin`, `gpioLastPin`	On chips that support GPIO functionality, these two fields identify the pins used by this device.

Description

A structure of this type is used to describe the capabilities of the audio output subsystem to the tmAO device library.

boardVIConfig_t

```
typedef struct{
   Char             codecName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t  (*init_func)(pboardVIParam_t params);
   tmLibdevErr_t  (*term_func)();
   tmLibdevErr_t  (*GetStandard)(
                    tmVideoAnalogStandard_t *standard);
   tmLibdevErr_t  (*setHue)(UInt val);
   tmLibdevErr_t  (*setSaturation)(UInt val);
   tmLibdevErr_t  (*setBrightness)(UInt val);
   tmLibdevErr_t  (*setContrast)(UInt val);
   tmLibdevErr_t  (*Configure)(UInt32 subaddr, UInt32 value);
   intInterrupt_t   intNumber;
   UInt32           mmioBase;
   UInt32           gpioFirstPin;
   UInt32           gpioLastPin;
   tmVideoCapabilitiesFlags_t  capFlags;
   tmVideoRGBYUVFormat_t       outputFormats;
   tmLibdevErr_t             (*getVSyncFallingEdge)(
                               pboardVIDec_t pVD,
                               UInt *lineNumber);
   tmLibdevErr_t             (*getSlicedData)(
                               pboardVIDec_t pVD,
                               UInt8 *Y,
                               UInt8 *U,
                               UInt8 *V,
                               tmVideoDataService_t service,
                               UInt size,
                               UInt8 *data,
                               UInt8 *dataSize);
   tmLibdevErr_t             (*getStatus)(
                               pboardVIDec_t pVD,
                               tmVideoStatusType_t type,
                               UInt *state);
   tmLibdevErr_t             (*getSupportedDataServices)(
                               tmVideoDataService_t fieldOne[],
                               tmVideoDataService_t fieldTwo[],
                               UInt8 tblSize);
   tmLibdevErr_t             (*setDataServices)(
                               pboardVIDec_t pVD,
                               tmVideoDataService_t fieldOne[],
                               tmVideoDataService_t fieldTwo[],
                               UInt8 tblSize);
   tmLibdevErr_t             (*enableSlicing)(
                               pboardVIDec_t pVD,
                               Bool enable);
   tmLibdevErr_t             (*setSlicerVideoStandard)(
                               pboardVIDec_t pVD,
                               tmVideoAnalogStandard_t standard);
```



```
tmLibdevErr_t           (*getSlicerVideoStandard)(
                          pboardVIDec_t pVD,
                          tmVideoAnalogStandard_t *standard);
tmLibdevErr_t           (*toggleFieldID)(
                          pboardVIDec_t pVD, Bool toggle);
tmLibdevErr_t           (*setSlicerInput)(
                          pboardVIDec_t pVD, UInt num);
tmLibdevErr_t           (*getSlicerInput)(
                          pboardVIDec_t pVD, UInt *num);
tmLibdevErr_t           (*setVideoColor)(
                          pboardVIDec_t pVD,
                          tmVideoColor_t color, UInt val);
tmLibdevErr_t           (*getVideoColor)(
                          pboardVIDec_t pVD,
                          tmVideoColor_t color, UInt *val);
tmLibdevErr_t           (*setAnalogInput)(
                          pboardVIDec_t pVD, UInt num);
tmLibdevErr_t           (*getAnalogInput)(
                          pboardVIDec_t pVD, UInt *num);
tmLibdevErr_t           (*setStandard)(
                          pboardVIDec_t pVD,
                          tmVideoAnalogStandard_t standard);
tmLibdevErr_t           (*setSourceType)(
                          pboardVIDec_t pVD,
                          tmVideoSourceType_t type);
tmLibdevErr_t           (*getSourceType)(
                          pboardVIDec_t pVD,
                          tmVideoSourceType_t *type);
tmLibdevErr_t           (*setOutputFormat)(
                          pboardVIDec_t pVD,
                          tmVideoRGBYUVFormat_t format);
tmLibdevErr_t           (*getOutputFormat)(
                          pboardVIDec_t pVD,
                          tmVideoRGBYUVFormat_t *format);
tmLibdevErr_t           (*setAcquisitionWnd)(
                          pboardVIDec_t pVD,
                          UInt beginX, UInt beginY,
                          UInt endX, UInt endY);
tmLibdevErr_t           (*getAcquisitionWnd)(
                          pboardVIDec_t pVD,
                          UInt *beginX, UInt *beginY,
                          UInt *endX, UInt *endY);
tmLibdevErr_t           (*getDefaultAcquisitionWnd)(
                          pboardVIDec_t pVD,
                          UInt *beginX, UInt *beginY,
                          UInt *endX, UInt *endY);
tmLibdevErr_t           (*setOutputSize)(
                          pboardVIDec_t pVD,
                          UInt width, UInt height);
tmLibdevErr_t           (*setInterlaceMode)(
```

```
                                pboardVIDec_t pVD, Bool interlace);
    tmLibdevErr_t              (*disableDecoder)(
                                pboardVIDec_t pVD, Bool disable);
    tmLibdevErr_t              (*enablePowerSaveMode)(
                                pboardVIDec_t pVD, Bool enable);
    tmLibdevErr_t              (*getGPIOCount)(
                                pboardVIDec_t pVD, UInt *num);
    tmLibdevErr_t              (*setGPIOState)(
                                pboardVIDec_t pVD,
                                UInt pin, Bool state);
    tmLibdevErr_t              (*getGPIOState)(
                                pboardVIDec_t pVD,
                                UInt pin, Bool *state);
    tmLibdevErr_t              (*openVBI)(
                                pboardVIDec_t pVD,
                                UInt sampleFreq, UInt startLine,
                                UInt numLines);
    tmLibdevErr_t              (*enableVBI)(
                                pboardVIDec_t pVD, Bool enable);
    tmLibdevErr_t              (*setVBIMode)(
                                pboardVIDec_t pVD,
                                tmVideoVBIMode_t mode);
    tmLibdevErr_t              (*setSlicerMode)(
                                pboardVIDec_t pVD,
                                tmVideoSlicerMode_t mode);
    tmLibdevErr_t              (*closeVBI)(
                                pboardVIDec_t pVD);
    tmLibdevErr_t              (*getSlicerLineFlags)(
                                pboardVIDec_t pVD,
                                Bool fieldOne[], Bool fieldTwo[],
                                UInt8 tblSize);
    boardVIDec_t                vDec;
} boardVIConfig_t;
```

Members

`codecName`	Name of the codec, in human-readable form.
`init_func`	Called from **viInstanceSetup** to configure board hardware. After successful completion, the video in hardware should be stopped and ready for action. Setting of device-specific MMIO registers and decoder-specific registers (over IIC, for example) happens here.
`term_func`	Called from **viClose**. Should shut down VI hardware.
`GetStandard`	Called from **viGetStandard** to retrieve current video standard.
`setHue`	Called from **viSetHue**.

`setSaturation`	Called from **viSetSaturation**.
`setBrightness`	Called from **viSetBrightness**.
`setContrast`	Called from **viSetContrast**.
`Configure`	A “backdoor” to support unforeseen features.
`standards`	OR’d supported standards.
`adapters`	OR’d supported adapter types
`intNumber`	Because multiple video inputs are supported, this field tells the software which interrupt to use.
`mmioBase`	Because multiple video inputs are supported, this field tells the software which set of MMIO registers to use.
`gpioFirstPin`	On chips that support GPIO functionality, this describes the first pin used by this device.
`gpioLastPin`	On chips that support GPIO functionality, this describes the first pin used by this device.
`capFlags`	Capabilities of connected video decoder device.
`outputFormats`	List of supported output formats.
`getVSyncFallingEdge`	Returns the line number where falling edge of vertical sync happens. This is important to determine the active video area.
`getSlicedData`	Returns sliced data according to requested service.
`getStatus`	Retrieves status information of the decoder chip.
`getSupportedDataServices`	Retrieves the decoder’s hardware slicing capabilities for each VBI line.
`setDataServices`	Sets the decoder’s hardware slicing for each VBI line.
`enableSlicing`	Enables the decoder’s hardware slicer.
`setSlicerVideoStandard`	Sets analog video standard for the decoder’s VBI slicer.
`getSlicerVideoStandard`	Returns current analog video standard used by the hardware slicer.
`toggleFieldID`	Toggles field ID to correct field assignment.
`setSlicerInput`	Sets the analog input for the VBI slicer.
`getSlicerInput`	Returns current analog input of the VBI slicer.
`setVideoColor`	Changes color settings for Brightness, Contrast, Saturation, or Hue.
`getVideoColor`	Retrieves value of current color settings for Brightness, Contrast, Saturation, or Hue.
`setAnalogInput`	Sets the analog input port.
`getAnalogInput`	Returns currently used analog input port.

`setStandard`	Sets the analog video standard.
`setSourceType`	Sets video source type: TV, video, or camera.
`getSourceType`	Retrieves current video source type settings.
`setOutputFormat`	Sets output format: YUV, RGB, ...
`getOutputFormat`	Retrieves current output format: YUV, RGB, ...
`setAcquisitionWnd`	Sets the video acquisition window.
`getAcquisitionWnd`	Retrieves the current acquisition window.
`getDefaultAcquisitionWnd`	Retrieves default window according to selected analog standard.
`setOutputSize`	Set output video size.
`setInterlaceMode`	Set the decoder's scaler to interlaced or field mode.
`disableDecoder`	Disables decoder, if supported.
`enablePowerSaveMode`	Enables the decoder's power save mode, if supported.
`getGPIOCount`	Returns the number of GPIO pins provided by the video decoder.
`setGPIOState`	Sets the state of a video decoder's GPIO pin.
`getGPIOState`	Returns current state of a video decoder's GPIO pin.
`openVBI`	Initializes VBI support of decoder for SW slicing.
`enableVBI`	Enables the decoder's VBI feature to support SW slicing.
`setVBIMode`	Sets the decoder's VBI mode.
`setSlicerMode`	Sets the decoder's VBI slicer mode.
`closeVBI`	Close the SW decoding of the VBI decoder.
`getSlicerLineFlags`	Get an indication for each VBI line if a specified data service was found or not.
`vDec`	Describes configuration data of the video decoder's instance. Currently it contains the decoder's IIC address, the number of supported video adapters and a list mapping the video adapters to input modes. It also contains instance information that the decoder uses internally.

Description

A structure of this type describes the capabilities of a video input subsystem of the TriMedia Video In Device Library.

boardVIParam_t

```
typedef struct {
   tmVideoAnalogStandard_t  videoStandard;
   tmVideoAnalogAdapter_t   adapterType;
   UInt32                   mmioBase;
   UInt                     adapterInstance;
} boardVIParam_t, *pboardVIParam_t;
```

Members

`videoStandard`	Defines the video analog standard to be expected on the video input interface. If set to **vasNone** the implementation tries to automatically detect the standard.
`adapterType`	Selects the adapter type to be CVBS or S-Video. If set to **vaaNone**, the default adapter will be used.
`mmioBase`	Because multiple video inputs are supported, this field tells the software which base address to use for the video input interface.
`adapterInstance`	If a board provides more then one CVBS or S-Video adapters this field selects the adapter's instance to use.

Description

A structure of this type is passed to the InstanceSetup function of the Video In device library to select a specific video input.

boardVIAdapterEntry_t

```
typedef struct {
   tmVideoAnalogAdapter_t  adapterType;
   UInt                    instNum;
   UInt                    decInput;
} boardVIAdapterEntry_t, *pboardVIAdapterEntry_t;
```

Members

`adapterType`	Selects the adapter type to be CVBS or S-Video.
`instNum`	Specifies an instance number for this adapter.
`decInput`	Selects the analog input mode mapping for the specified adapter and instance.

Description

Structure mapping video input adapters to decoder inputs.

boardVIDec_t

```
typedef struct {
   UInt                     slaveAddr;
   UInt                     numAdapters;
   pboardVIAdapterEntry_t   adapterTable;
   tmVideoAnalogStandard_t  currentStandard;
} boardVIDec_t, *pboardVIDec_t;
```

Members

`slaveAddr`	IIC slave address of the decoder.
`numAdapters`	Number of supported adapters.
`adapterTable`	Points to a table mapping the adapters instances to decoder inputs.
`currentStandard`	Stores the current video standard. Only used internally

Description

Structure describes configuration data of the video decoder's instance. It contains the decoder's IIC address, the number of supported video adapters and a list mapping the video adapters to input modes. It also contains instance information that the decoder uses internally.

boardVOConfig_t

```
typedef struct{
   Char             codecName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t  (*init_func    )( pboardVOParam_t params );
   tmLibdevErr_t  (*term_func    )( void );
   tmLibdevErr_t  (*setHue       )( UInt val );
   tmLibdevErr_t  (*setSaturation)( UInt val );
   tmLibdevErr_t  (*setBrightness)( UInt val );
   tmLibdevErr_t  (*setContrast  )( UInt val );
   tmLibdevErr_t  (*Configure    )( UInt32 subaddr, UInt32 value );
   UInt32           standards;
   UInt32           adapters;
   intInterrupt_t   intNumber;
   UInt32           mmioBase;
} boardVOConfig_t;
```

Fields

`codecName`	Name of the codec in human-readable format.
`init_func`	Called from **voInstanceSetup** to configure the board hardware. After successful completion, the video-out hardware should be ready for action of device specific MMIO board and/or decoder specific register initialization.
`term_func`	Called from **voClose**. It should shut down the video-out hardware.
`SetHue`	Called from **voSetHue**. It sets the hue of the video-out encoder to value: **val**.
`SetSaturation`	Called from **voSetSaturation**. It sets the saturation of the video-out encoder to value: **val**.
`SetBrightness`	Called from **viSetBrightness**. It sets the brightness of the video-out encoder to value: **val**.
`SetContrast`	Called from **viSetContrast**. It sets the contrast of the video-out encoder to value: **val**.
`Configure`	This is a backdoor to support features not foreseen in the initial design.
`standards`	This is a bitmask of OR'ed **tmVideoAnalogStandard_t** supported by the codec. It is initialized by the board's init function and is reported by **voGetCapabilities**.
`adapters`	This is an OR'd bitmask of type **tmVideoAnalogStandard_t** supported by the codec. It is initialized by the board's init function and is reported by **voGetCapabilities**.

Description

A structure of this type describes the capabilities of the video output subsystem of the TriMedia Video-Out Device Library.

boardSSIParam_t

```
typedef struct {
   UInt8   interruptLevelSelect;
   void   *configBuffer;
   UInt32  configBufferLength;
} boardSSIParam_t;
```

Fields

`interruptLevelSelect`	This value is passed in with **ssiInstanceSetup**, it is the interrupt level at which the interrupt service routine gets invoked.
`configBuffer`	This is a pointer to a buffer for anything else the board code needs.
`configBufferLength`	The buffer length.

Description

This is a parameter structure for communication between the telecom AFE and SSI device library. A structure of this type is passed to the SSI's init_func by **ssiInstanceSetup**.

boardSSIConfig_t

```
typedef struct{
   Char                        afeName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t              (*init_func)( boardSSIParam_t *param );
   tmLibdevErr_t              (*termFunc) ( void );
   tmLibdevErr_t              (*afe_Hook) ( Bool offHook );
   mLibdevErr_t               (*Configure)( boardSSIParam_t *param );
   tmSSIAnalogConnection_t     connectionFlags;
   void                       *reserved;
} boardSSIConfig_t;
```

Fields

`afeName`	This is the analog front end's name in human-readable text.
`initFunc`	This function is called in **ssiInstanceSetup**. It sets any non-default values of MMIO registers, takes the AFE out of reset (using IIC, for example) and leaves the AFE stopped and ready. Because an AFE like the 7545 cannot be left in "stopped" mode, configuration for the st7545 is left to an application.
`termFunc`	This function is called in **ssiClose**. It will leave the SSI hardware shut down.
`afe_Hook`	Currently reserved. Hook control is left to the application.
`Configure`	This is a backdoor to support features not foreseen in the initial design.
`connectionFlags`	This is used to report a list of supported connections in **ssiGetCapabilities**.
`reserved`	Reserved for future expansion.

Description

This structure is used in the **boardConfig_t** to describe the telecom interface on board.

boardSPDOParam_t

```
typedef struct {
   tmAudioTypeFormat_t     audioTypeFormat;
   UInt32                  audioSubtypeFormat;
   UInt32                  audioDescription;
   Float                   sRate;
   Int                     size;
   tmAudioAnalogAdapter_t  output;
} boardSPDOParam_t, *pboardSPDOParam_t;
```

Fields

`audioTypeFormat`	Audio type.
`audioSubtypeFormat`	Audio subtype.
`audioDescription`	Additional description of the audio data.
`sRate`	Sample rate in Hz.
`size`	Size of buffers, in samples.
`output`	Output.

Description

Structures of this type are used by the tmSPDO device library to initialize the board component of the S/PDIF out system.

boardSPDOConfig_t

```
typedef struct{
   Char              codecName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t   (*initFunc)  ( pboardSPDOParam_t setup );
   tmLibdevErr_t   (*termFunc)  ( void );
   tmLibdevErr_t   (*startFunc) ( void );
   tmLibdevErr_t   (*stopFunc)  ( void );
   tmLibdevErr_t   (*setSRate)  ( Float sRate );
   tmLibdevErr_t   (*getSRate)  ( Float *sRate );
   tmLibdevErr_t   (*configFunc)( UInt32 subAddr, Pointer value );
   UInt32            audioTypeFormats;
   UInt32            audioSubtypeFormats;
   UInt32            audioAdapters;
   intInterrupt_t    intNumber;
   UInt32            mmioBase;
   Float             maxSRate;
   Float             minSRate;
} boardSPDOConfig_t;
```

Fields

`codecName`	Codec name, in human-readable format.
`initFunc`	Function called in **spdoInstanceSetup**. On success, must leave the S/PDIF output system "stopped" but otherwise ready for action. MMIO setup will be done, as well as other board or codec-specific actions, such as the setting of IIC control bits or the initialization of codec registers.
`termFunc`	Called in **spdoClose**. Should leave S/PDIF output system shut down.
`startFunc`	Called from **spdoStart**.
`stopFunc`	Called from **spdoStop**.
`setSRate`	Called from **spdoSetSRate**.
`getSRate`	Called from **spdoGetSRate**. Should return an accurate value from the hardware.
`configFunc`	Reserved to support features not forseen in the initial design.

Description

A structure of this type is used to describe the capabilities of the S/PDIF output subsystem to the tmSPDO device library.

boardTPConfig_t

```
typedef struct {
   Char              codecName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t   (*initFunc)   ( void );
   tmLibdevErr_t   (*termFunc)   ( void );
   tmLibdevErr_t   (*configFunc) ( UInt32 address, UInt32 value );
   Bool              onlySampleDvalidBytes;
   Bool              sampleOnNegativeEdge;
   intInterrupt_t    intNumber;
   UInt32            mmioBase;
} boardTPConfig_t;
```

Fields

`codecName`	Codec name, in human-readable format.
`initFunc`	Initialization function.

Description

Describes and controls the Transport Block on TM-2*xxx* processors.

hdvoImageOutputMode_t

```
typedef enum {
   /* the following are YUV only */
   hdvoImageOutput_8b_YUV422   = 0x00000001,
   hdvoImageOutput_10b_YUV422  = 0x00000002,
   hdvoImageOutput_16b_YUV422  = 0x00000004,
   hdvoImageOutput_20b_YUV422  = 0x00000008,
   hdvoImageOutput_24b_YUV422  = 0x00000010,
   hdvoImageOutput_30b_YUV422  = 0x00000020,
   /* the following can be RGB or YUV */
   hdvoImageOutput_8b_444      = 0x00000040,
   hdvoImageOutput_10b_444     = 0x00000080,
   hdvoImageOutput_24b_444     = 0x00000100,
   hdvoImageOutput_30b_444     = 0x00000200
} hdvoImageOutputMode_t;
```

Description

Describes and controls the HDVO Block on TM-2xxx processors.

boardHDVOConfig_t

```
typedef struct {
   Char                  codecName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t        (*initFunc)(
                         tmVideoAnalogStandard_t  standard,
                         tmVideoAnalogAdapter_t   adapterType,
                         tmVideoTypeFormat_t      videoType,
                         hdvoImageOutputMode_t    videoSubtype,
                         UInt32                   description );
   tmLibdevErr_t        (*termFunc)( void );
   tmLibdevErr_t        (*configFunc)(
                         UInt32  address,
                         UInt32  value );
   UInt32                standards;
   UInt32                adapters;
   tmVideoTypeFormat_t   videoType;
   hdvoImageOutputMode_t videoSubtype;
   UInt32                description;
   intInterrupt_t        intNumber;
} boardHDVOConfig_t;
```

Fields

`codecName`	Name of the codec in human-readable format.
`initFunc`	Initialization function.
`standards`	OR'd supported standards.
`adapters`	OR'd supported adapter types.
`videoType`	OR'd supported video types.
`videoSubtype`	OR'd supported video subtypes.
`description`	OR'd supported video flags.
`intNumber`	Number of the interrupt.

Description

Describes and controls the HDVO Block on TM-2*xxx* processors.

boardConfig_t *(obsolete)*

```
typedef struct{
   boardAIConfig_t   *aibc;
   boardAOConfig_t   *aobc;
   boardVIConfig_t   *vibc;
   boardVOConfig_t   *vobc;
   boardSSIConfig_t  *ssibc;
   void              (*board_init_func  )( void );
   Int32             (*board_detect_func)( void );
   void               *reserved;
   UInt32              ID;
} boardConfig_t;
```

Fields

`aibc`	Pointer to the description of the audio input.
`aobc`	Pointer to the description of the audio output.
`vibc`	Pointer to the description of the video input.
`vobc`	Pointer to the description of the video output.
`ssibc`	Pointer to the description of the synchronous serial port connections.
`board_init_func`	This function is called to initialize the board peripherals, and then to take the board out of reset.
`board_detect_func`	This function returns zero if the actual hardware matches this board description. The hardware should be identified using the subsystem ID contained in the EEPROM and copied to the PCI configuration field.
`reserved`	Reserved for future expansion.
`ID`	The ID field is uniquely defined by the programmer implementing the board support package. It should be entered in tmBoardID.h. That file contains some guidelines for the choice of an ID.

Description

From the perspective of the device libraries, a structure of **boardConfig_t** completely describes the off chip hardware. A board support file (such as philips_iref.c) must be created and for each supported board. Default versions are contained in the standard libdev.a. You can override the standard configuration in his local code, or you can replace the structures in libdev.a.

Important
This structure should not be used in new code and is being maintained for compatibility purposes only. Instead, use **tsaBoardGet*** functions to retrieve this information.

boardPICIntCaps_t

```
typedef struct boardPICIntCaps {
   intInterrupt_t   interrupt;
   Bool             levelTriggered;
   intPriority_t    priority;
   UInt32           numSources;
   tsaPICSource_t  *sourceTable;
   tmLibdevErr_t  (*initFunc        )( UInt32 source );
   tmLibdevErr_t  (*termFunc        )( UInt32 source );
   Bool           (*sourceDetectFunc)( UInt32 *source );
   tmLibdevErr_t  (*startFunc       )( UInt32 source );
   tmLibdevErr_t  (*stopFunc        )( UInt32 source );
   tmLibdevErr_t  (*ackFunc         )( UInt32 source );
} boardPICIntCaps_t, *pboardPICIntCaps_t;
```

Fields

`interrupt`	TriMedia interrupt number used for this interrupt.
`levelTriggered`	Indicates in which mode the interrupt must be used by the PIC device library. True: level triggered, False: edge triggered.
`priority`	Priority that must be used for the interrupt.
`numSources`	Number of sources that are supported by this interrupt.
`sourceTable`	Array (size: numSources) that contains the sources that are supported by the interrupt.
`initFunc`	This function is called in **tsaPICInstanceSetup**. It does the basic initiallization of the external hardware for this interrupt. On success it must leave the hardware in a state that interrupt sources can be enabled.
`termFunc`	This function gets called in **tsaPICClose.** It should shut down the external hadware used by the interrupt.
`sourceDetectFunc`	This function is called in the PIC interrupt service routine. If it returns True it detected the source for the interrup and writes the index of the detected source in **sourceTable** to the function's parameter **source**. In the case that it returns False there is no more pending interrupt source.
`startFunc`	Enables the selected interrupt source. The parameter **source** is the index to the source in **sourceTable**.

`stopFunc`	Disables the selected interrup source. The parameter **source** is the index to the source in **sourceTable**.
`ackFunc`	Acknowledges the interrupt source. The parameter **source** is the index to the source in **sourceTable**.

Description

An array of structures of this type is used in **boardPICConfig_t** to describe the capabilities of the supported interrupts to the PIC device library.

boardPICConfig_t

```
typedef struct boardPICConfig {
   Char                picName[DEVICE_NAME_LENGTH];
   UInt32              numSupportedInterrupts;
   pboardPICIntCaps_t  intCaps;
} boardPICConfig_t, *pboardPICConfig_t;
```

Fields

`picName`	Name for the PIC hardware.
`numSupportedInterrupts`	Number of supported interrupts.
`intCaps`	Array of interrupt capabilities.

Description

This structure describes capabilities of the interrupts supported on the board.

boardUartParam_t

```
typedef struct {
   tsaUartBaud_t    baudRate;
   Int              numDataBits;
   Int              numStopBits;
   tsaUartParity_t  parity;
   Bool             enableModemInt;
}boardUartParam_t, *pboardUartParam_t;
```

Fields

`baudRate`	Baudrate.
`numDataBits`	Number of data bits.
`numStopBit`	Number of stop bits.
`parity`	Parity mode.
`enableControlInt`	This flag indicates if the control interrupt must be enabled. True: enable control interrupt, False: don't enable control interrupt.

Description

A structure of this type is passed to the UART init function and describes the required setup.

boardUartConfig_t

```
typedef struct {
   Char              name[DEVICE_NAME_LENGTH];
   tmLibdevErr_t   (*initFunc)( unitSelect_t portID,
                     pboardUartParam_t param );
   tmLibdevErr_t   (*termFunc)( unitSelect_t portID );
   tmLibdevErr_t   (*readDataFunc)( unitSelect_t portID,
                     Address data );
   tmLibdevErr_t   (*setTxIntFunc)( unitSelect_t portID,
                     Bool enable );
   tmLibdevErr_t   (*writeDataFunc)( unitSelect_t portID,
                     Char data );
   tmLibdevErr_t   (*setRxIntFunc)( unitSelect_t portID,
                     Bool enable );
   Bool            (*getEventFunc)( unitSelect_t portID,
                     UInt32 *event );
   tmLibdevErr_t   (*configFunc)( unitSelect_t portID,
                     UInt32 command, Pointer value );
   UInt32            baudRates;
} boardUartConfig_t, *pboardUartConfig_t;
```

Fields

`name`	Name of the UART.
`initFunc`	Called by the **tsaUartInstanceSetup** function. On success it should leave the hardware in a state so that the application can start to transmit or receive data.
`termFunc`	Called by **tsaUartClose** to shut down the UART hardware for the selected port.
`readDataFunc`	Called by the UART device library to read one received character.
`setTxIntFunc`	Called by the UART device library to enable/disable the transmit interrupt.
`writeDataFunc`	Called by the UART device library to send one character.
`setRxIntFunc`	Called by the UART device library to enable/disable the receive interrupt.
`getEventFunc`	Called by the UART interrupt service routine to get the event that triggered the interrupt.
`configFunc`	Used in the UART device library to configure the UART.
`baudRates`	Supported baudrates.

Description

A structure of this type is used to describe the capabilities of the UART hardware.

boardIRParam_t

```
typedef struct{
   tsaIRDevice_t   device;
} boardIRParam_t, *pboardIRParam_t;
```

Fields

`device`	Device type that wil be used.

Description

A structure of this type is passed to the flash init function and describes the required setup.

boardIRConfig_t

```
typedef struct {
   Char             irName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t  (*initFunc    )( pboardIRParam_t params );
   tmLibdevErr_t  (*termFunc    )( void );
   tmLibdevErr_t  (*startFunc   )( void );
   tmLibdevErr_t  (*stopFunc    )( void );
   tmLibdevErr_t  (*configFunc  )( UInt32 command, Pointer value );
   void           (*getEventFunc)( tsaIREvent_t *event,
                    UInt32* value );
   UInt32           supportedDevices;
} boardIRConfig_t, * pboardIRConfig_t;
```

Fields

`irName`	Name of the IR hardware.
`initFunc`	This function gets called in the **tsaIRInstanceSetup** function and on success should leave the IR hardware in a state that IR reception can be started.
`termFunc`	This function gets called in **tsaIRClose** it shuts down the IR hardware.
`startFunc`	Gets called in **tsaIRStart** and starts IR reception.
`stopFunc`	Gets called in **tsaIRStop** and stops infrared reception.
`configFunc`	Gets called in **tsaIRConfig**. This function can be used to configure the IR device. It can also be used to support features not implemented in the current IR device library since the config command and value are passed directly to this function without interpretation in the device library.
`getEventFunc`	This function gets called in the device library's interrupt handler to get the event that caused the IR interrupt.
`supportedDevices`	Supported IR devices.

Description

This structure describes capabilities of the IR hardware supported on the board.

boardFlashConfig_t

```
typedef struct boardFlashConfig{
   Char              flashName[DEVICE_NAME_LENGTH];
   tmLibdevErr_t   (*initFunc)( void );
   tmLibdevErr_t   (*readWordFunc) (UInt32 address, UInt32 *data );
   tmLibdevErr_t   (*writeWordFunc)(UInt32 address, UInt32 data );
   tmLibdevErr_t   (*readBlockFunc)(UInt32 address, UInt32 *data,
                     UInt32 numberOfWords);
   tmLibdevErr_t   (*writeBlockFunc)(UInt32 address, UInt32 *data,
                     UInt32 numberOfWords);
   tmLibdevErr_t   (*eraseSectorFunc)(UInt32 sectorNumber);
   tmLibdevErr_t   (*eraseAllFunc)( void );
   UInt32            flashSize;
   UInt32            numberOfSectors;
   UInt32            sectorSize;
   UInt32            erasedWord;
} boardFlashConfig_t, *pboardFlashConfig_t;
```

Fields

`flashName`	Name of the flash hardware.
`initFunc`	This function initializes the flash. It can only be called once. If it returns an error the flash might allready be in used by another component and can not be used by the current caller (e.g. flash file system and flash device library are sharing the same board interface).
`readWordFunc`	Read one word from flash.
`writeWordFunc`	Write one word to flash.
`readBlockFunc`	Read a block of words from flash.
`writeBlockFunc`	Write a block of words to flash.
`eraseSectorFunc`	Erase a flash sector.
`eraseAllFunc`	Erase the complete flash.
`flashSize`	Flash size.
`numberOfSectors`	Number of flash sectors.
`sectorSize`	Size of one flash sector.
`erasedWord`	Value that you read from flash after it has been erased.

Description

This structure describes capabilities of the flash hardware supported on the board.

Note: The size of one word is 32 bits (4 bytes). All addresses and size values are based on a 32-bit word.

TMBoard API Functions

This section presents the TMBoard API functions.

A. These functions should not be used in new code and are being maintained for compatibility purposes only. Use **tsaBoardGet*** instead.

tsaBoardRegisterAO

```
extern tmLibdevErr_t tsaBoardRegisterAO(
   UInt32            unitNumber,
   boardAOConfig_t  *AOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`AOconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterSPDO

```
extern tmLibdevErr_t tsaBoardRegisterSPDO(
   UInt32              unitNumber,
   boardSPDOConfig_t  *SPDOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`SPDOconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterAI

```
extern tmLibdevErr_t tsaBoardRegisterAI(
   UInt32            unitNumber,
   boardAIConfig_t  *AIconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`AIconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterVO

```
extern tmLibdevErr_t tsaBoardRegisterVO(
   UInt32           unitNumber,
   boardVOConfig_t *VOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`VOconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterVI

```
extern tmLibdevErr_t tsaBoardRegisterVI(
   UInt32            unitNumber,
   boardVIConfig_t  *VIconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`VIconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterSSI

```
extern tmLibdevErr_t tsaBoardRegisterSSI(
   UInt32             unitNumber,
   boardSSIConfig_t  *SSIconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`SSIconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
TMLIBDEV_OK	Success.

tsaBoardRegisterTP

```
extern tmLibdevErr_t tsaBoardRegisterTP(
   UInt32            unitNumber,
   boardTPConfig_t  *TPconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`TPconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterHDVO

```
extern tmLibdevErr_t tsaBoardRegisterHDVO(
   UInt32              unitNumber,
   boardHDVOConfig_t  *HDVOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`HDVOconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterGPIO

```
extern tmLibdevErr_t tsaBoardRegisterGPIO(
   UInt32              unitNumber,
   boardGPIOConfig_t  *GPIOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions.
`GPIOconfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit has already been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterPIC

```
extern tmLibdevErr_t tsaBoardRegisterPIC(
   UInt32              unitNumber,
   boardPICConfig_t  *picConfig
);
```

Parameters

`unitNumber`	This corresponds with the unit number which is specified with the **devOpenM** function.
`picConfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h or tsaDevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit already has been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterUart

```
extern tmLibdevErr_t tsaBoardRegisterUart(
   UInt32              unitNumber,
   boardUartConfig_t  *uartConfig
);
```

Parameters

`unitNumber`	This corresponds with the unit number which is specified with the **devOpenM** function.
`uartConfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h or tsaDevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit already has been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterIR

```
extern tmLibdevErr_t tsaBoardRegisterIR(
   UInt32            unitNumber,
   boardIRConfig_t  *irConfig
);
```

Parameters

`unitNumber`	This corresponds with the unit number which is specified with the **devOpenM** function.
`irConfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h or tsaDevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit already has been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterFlash

```
extern tmLibdevErr_t tsaBoardRegisterFlash(
   UInt32               unitNumber,
   boardFlashConfig_t  *flashConfig
);
```

Parameters

`unitNumber`	This corresponds with the unit number which is specified with the **devOpenM** function.
`flashConfig`	Pointer to the board configuration of this device. These structures are defined in the tmdevboard.h or tsaDevboard.h headers. This pointer must point to a static variable or to a malloc'd variable, since only the value of the pointer is copied in the registry, and not the content of the structure itself.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TSA_BOARD_ERR_UNIT_EXISTS`	The unit already has been registered.
`TMLIBDEV_OK`	Success.

tsaBoardRegisterBoard

```
extern tmLibdevErr_t tsaBoardRegisterBoard(
   UInt32   ID,
   Char    *boardName
);
```

Parameters

`ID`	Board ID, as read from the EEPROM of the board.
`boardName`	Pointer to a null-terminated string describing the board. The string length should be 32 characters or less (e.g.,"Philips Iref").

Return Codes

`TMLIBDEV_OK`	Success
`TSA_BOARD_ERR_BOARD_NAME_TOO_LONG`	The board name is too long (more than 32 characters.

Description

Creates two new entries in the registry: bsp/boardID and bsp/boardName. These entries contain the parameters **ID** and **boardName**. The ID number and the boardName need not be statically allocated or malloc'd. The registry makes its own copy.

tsaBoardGetAO

```
extern tmLibdevErr_t tsaBoardGetAO(
   UInt32             unitNumber,
   boardAOConfig_t  **AOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`AOconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetSPDO

```
extern tmLibdevErr_t tsaBoardGetSPDO(
   UInt32               unitNumber,
   boardSPDOConfig_t  **SPDOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`SPDOconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered read-only, since it actually points to the system value of the structure.

tsaBoardGetAI

```
extern tmLibdevErr_t tsaBoardGetAI(
   UInt32             unitNumber,
   boardAIConfig_t  **AIconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`AIconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetVO

```
extern tmLibdevErr_t tsaBoardGetVO(
   UInt32             unitNumber,
   boardVOConfig_t  **VOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`VOconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetVI

```
extern tmLibdevErr_t tsaBoardGetVI(
   UInt32             unitNumber,
   boardVIConfig_t  **VIconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`VIconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetSSI

```
extern tmLibdevErr_t tsaBoardGetSSI(
   UInt32              unitNumber,
   boardSSIConfig_t  **SSIconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`SSIconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetTP

```
extern tmLibdevErr_t tsaBoardGetTP(
   UInt32             unitNumber,
   boardTPConfig_t  **TPconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`TPconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetHDVO

```
extern tmLibdevErr_t tsaBoardGetHDVO(
   UInt32               unitNumber,
   boardHDVOConfig_t  **HDVOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`HDVOconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetGPIO

```
extern tmLibdevErr_t tsaBoardGetGPIO(
   UInt32               unitNumber,
   boardGPIOConfig_t  **GPIOconfig
);
```

Parameters

`unitNumber`	This corresponds to the unit number which is specified with the **devOpenM** functions
`GPIOconfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tmdevboard.h headers.

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The passed **unitNumber** is not in the valid range of unit numbers (0–99), or when there is no registered device for this unit.
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetPIC

```
extern tmLibdevErr_t tsaBoardGetPIC(
   UInt32              unitNumber,
   boardPICConfig_t  **picConfig
);
```

Parameters

`unitNumber`	This corresponds with the unit number which is specified with the **devOpenM** function.
`picConfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tsaDevBoard.h headers (or tmdevboard.h).

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered read-only, because it actually points to the system value of the structure.

tsaBoardGetUart

```
extern tmLibdevErr_t tsaBoardGetUart(
   UInt32               unitNumber,
   boardUartConfig_t  **uartConfig
);
```

Parameters

`unitNumber`	This corresponds with the unit number which is specified with the **devOpenM** function.
`uartConfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tsaDevBoard.h headers (or tmdevboard.h).

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetIR

```
extern tmLibdevErr_t tsaBoardGetIR(
   UInt32             unitNumber,
   boardIRConfig_t  **irConfig
);
```

Parameters

`unitNumber`	This corresponds with the unit number which is specified with the **devOpenM** function.
`irConfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tsaDevBoard.h headers (or tmdevboard.h).

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered read-only, because it actually points to the system value of the structure.

tsaBoardGetFlash

```
extern tmLibdevErr_t tsaBoardGetFlash(
   UInt32                unitNumber,
   boardFlashConfig_t  **flashConfig
);
```

Parameters

`unitNumber`	This corresponds with the unit number which is specified with the **devOpenM** function.
`flashConfig`	Pointer to a pointer to a **boarddevConfig_t** structure. This structure is defined in the tsaDevBoard.h headers (or tmdevboard.h).

Return Codes

`TSA_BOARD_ERR_INVALID_UNIT_NUMBER`	The unit number is not in a valid range (0–99).
`TMLIBDEV_OK`	Success.

Description

The returned structure should be considered as read-only, since this actually points to the system value of the structure.

tsaBoardGetBoard

```
extern tmLibdevErr_t tsaBoardGetBoard(
   UInt32  *pID,
   Char   **pboardName
);
```

Parameters

pID	Pointer to a buffer that receives the board ID.
pboardName	Pointer to a pointer to a string that receives the board name.

Return Codes

TMLIBDEV_OK	Success.
TSA_BOARD_ERR_GETBOARDCONFIG_FAILED	The function was unable to read the board ID or the boardName from the registry. **pID**, and **pboardName** get assigned default values.

Description

Returns, in **pID** and in **pBoardName**, the board ID and a small string describing the actual board (e.g., "Philips Iref"). The function reads these values from the entries bsp/boardID and bsp/boardName in the registry. This assumes that a call to **tsaBoardRegisterBoard** was successful. The **boardName** variable should be considered read only, because the actual returned value points directly to the content of the registry.

boardGetConfig *(obsolete)*

```
tmLibdevErr_t boardGetConfig(
   boardConfig_t   *bc
)
```

Parameters

bc	Pointer to the board configuration struct to fill.

Return Codes

TMLIBDEV_OK	Success.
BOARD_ERR_NULL_DETECT_FUNCTION	One of the boards defined in boardcfg.c did not have a detect function.

Description

This function is called by the device libraries in order to retrieve pointers to the data used for initialization and configuration. On successful completion, the parameter bc points to a board config structure which describes this board. Application code should not have to call this function, as the device library should handle all access to the board support package.

Operational Details: if the board has not yet been initialized, step through the **_board_config_array** calling each **board_detect_func** in turn until one of them returns zero. The **board_init_func** for this board is then called. Finally, the address of the boardConfig_t structure for the detected board is returned. If none of the detect functions succeed, the first one in the list is returned as a default.

IMPORTANT
This function should not be used in new code and is being maintained for compatibility purposes only. Use **tsaBoardGet*** instead.

boardGetID *(obsolete)*

```
tmLibdevErr_t boardGetID(
   UInt   *ID
);
```

Parameters

`ID`	Pointer to variable to be filled with ID value.

Return Codes

`TMLIBDEV_OK`	Success.
`BOARD_ERR_NULL_DETECT_FUNCTION`	One of the boards defined in boardcfg.c did not have a detect function.

Description

This function is called by an application in order to retrieve the ID of the current board. Valid IDs are defined in tmBoardID.h.

If the board has not yet been initialized, step through the _board_config_array calling each **board_detect_func** in turn until one of them returns zero. The **board_init_func** for this board is then called. Finally, the contents of the ID field in the **boardConfig_t** for the detected board is returned. If none of the detect functions succeed, the ID of the first board in the list is returned as a default.

IMPORTANT
This function should not be used in new code and is being maintained for compatibility purposes only. Use **tsaBoardGet*** instead.

Chapter 20

Exceptions API

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

Overview

An exception (for example, DBZ and OVF) is special-event processing which occurs under the conditions described in the appropriate TriMedia data book (Section 3.4.2, EXC (Exceptions)).

Exceptions are modeled according to the general TriMedia device model. That is, each exception is considered an "instance" which must be "opened" before it can be used, and must be "closed" when no longer used, to make it available for other software components.

Opening an exception means that the requested exception is reserved for use by the caller. A failure to open indicates that particular exception is currently in use, and therefore reserved, by other software. *Using* an exception means that an exception handler is provided, together with an enabling flag, so that later exceptions can be caught and processed.

New values for the "instance setup" parameters can be set using the function **excInstanceSetup**, while the ones currently in use can be obtained using function **excGetInstanceSetup**.

The handler for a particular exception may be any C function with a prototype according to type **excHandler** (see type definitions), or NULL. Contrary to interrupt handlers, exception handlers must not be compiled with a handler pragma. (Do not use the **TCS_exception_handler**.) In case the current handler for a particular exception is not equal to NULL, the enable flag fully determines whether the corresponding exception instance is enabled in the PCSW. If the flag is equal to TRUE, an occurrence of the exception will cause the handler to be called with the parameter values **dpc** and **spc** as described in the TriMedia data book(s), after clearing the exception's pending flag. If the flag is equal to FALSE, the handler is installed, but exceptions are kept pending until it is enabled by a later call to **excInstanceSetup**. The handler may also be NULL. In this case, the enabled flag is overruled, and the corresponding exception is not enabled. This can be used for keeping a particular exception reserved without actually installing a handler.

Note
Integer division is implemented with a call to the floating point division hardware. This may raise the sticky "INX" exception bit. Integer division neither uses nor clears this bit. This issue must be addressed by any developer wishing to use the floating point exceptions.

The following can be used to install and enable a divide-by-zero handler:

```
#include "tm1/tmExceptions.h"
static void DBZHandler(UInt32 dpc, UInt32 spc){
   printf("DBZHandler: zero divide\n");
}
excInstanceSetup_t setup;

setup.enabled = True;
setup.handler = DBZHandler;
```



```
if( excOpen(excDBZ)                 != 0  ) { error(); }
if( excInstanceSetup( excDBZ, &setup != 0 )) { error(); }
```

See also the full example in $(TCS)/examples/peripherals/exceptions.

IMPORTANT
Contrary to interrupt handlers, exception handlers must NOT be compiled using any particular pragma. tmException contains an internal exception handler (a "real one") which dispatches the user provided handlers, calling them as "normal" C functions.

Note
The exceptions are automatically acknowledged before calling the handler, and their pending flags are cleared.

Exceptions API Data Structures

The following sections present the Exceptions API device library data structures. They are contained in the file tmExceptions.h.

excHandler

```
typedef void (*excHandler) (
   UInt32   dpc,
   UInt32   spc
);
```

Fields

`dpc`	Destination Program Counter. The intended destination address of the successful jump. (See TriMedia data book(s), Sec. 3.4.2, EXC (Exceptions)).
`spc`	Source Program Counter.

Description

This is a callback function prototype for an exception handler. It is used as a field in the struct **excInstanceSetup_t**.

excException_t

```
typedef enum{
   excDBZ =  Ø,
   excINX =  1,
   excUNF =  2,
   excOVF =  3,
   excINV =  4,
   excIFZ =  5,
   excOFZ =  6,
   excTFE = 26,
   excRSE = 13,
   excWBE = 14,
   excMSE = 15
} excException_t;
```

Description

This enum is used to select the particular exception. It is used as a field in the struct **excInstanceSetup_t**, and as a parameter in the functions **excGetInstanceSetup**, **excOpen** and **excClose**.

excCapabilities_t

```
typedef struct excCapabilities_t{
   tmVersion_t  version;
   Int          numSupportedInstances;
   Int          numCurrentInstances;
} excCapabilities_t, *excCapabilities_t;
```

Fields

`version`	This allows the application software to identify and support possible changes to the interface.
`numSupportedInstances`	Number of supported instances, currently 16, one for each exception handled.
`numCurrentInstances`	Number of instances open, incremented when a new instance is allocated, decremented as each is closed.

Description

This struct will hold the capabilities data of the Exception Handler. It is used as a parameter for the function **excGetCapabilities**.

excInstanceSetup_t

```
typedef struct {
   Bool         enabled;
   excHandler   handler;
} excInstanceSetup_t, *pexcInstanceSetup_t;
```

Fields

`enabled`	Control flag.
`handler`	Installed handler, or NULL.

Description

This struct provides the handler addresses and flag control. It is used as a parameter in the functions **excInstanceSetup** and **excGetInstanceSetup**.

Exceptions API Functions

This section presents the Exceptions API device library functions.

excGetCapabilities

```
tmLibdevErr_t excGetCapabilities(
   pexcCapabilities_t   *cap
);
```

Parameters

`cap`	Pointer to the struct of type **excCapabilities_t** where the data is placed.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts, in the debug version of the device library, when **cap** is NULL.

Description

The function **excGetCapabilities** is used to retrieve global capabilities from a static variable.

excInstanceSetup

```
tmLibdevErr_t excInstanceSetup(
   excException_t       instance,
   excInstanceSetup_t  *setup
);
```

Parameters

`instance`	Instance is an enum used to select the particular exception.
`setup`	Pointer to a struct holding handler pointer and control flag.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NOT_OWNER`	Is asserted by the debug version of the device library if instance does not match the owner or if an incorrectly sized struct is passed.

Description

The function **excInstanceSetup** is used to set and change instance parameters. It disables interrupts, copies the value of setup in a static array variable (for internal use only), restores interrupts.

excGetInstanceSetup

```
tmLibdevErr_t excGetInstanceSetup(
   excException_t       Instance,
   excInstanceSetup_t  *setup
);
```

Parameters

`instance`	Instance is an enum used to select the particular exception.
`setup`	Pointer to buffer receiving returned parameters.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NULL_PARAMETER`	Is asserted by the debug version of the device library if setup is NULL.
`TMLIBDEV_ERR_NOT_OWNER`	Is asserted by the debug version of the device library if instance does not match the owner.

Description

The function retrieves instance parameters.

excOpen

```
tmLibdevErr_t excOpen(
   excException_t   instance
);
```

Parameters

`instance`	Instance is an enum used to select the particular exception.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NULL_PARAMETER`	Is asserted by the debug version of the device library if instance is NULL.

Description

The function **excOpen** is used to reserve the use of a specified exception.

excClose

```
tmLibdevErr_t excClose(
   excException_t   instance
);
```

Parameters

`instance`	Instance is an enum used to select the particular exception.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NOT_OWNER`	Is asserted by the debug version of the device library if instance does not match the owner.

Description

The function deallocates the exception instance and uninstalls its handler when it has one.

Chapter 21

TriMedia Interrupts API

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

Overview

Interrupts are modeled according to the general TriMedia device model, that is, each interrupt (e.g. intAUDIOIN, intIIC) is considered an instance which must be opened before it can be used, and must be closed when no longer used, to make it available for other software components. Opening either succeeds, which means that the requested interrupt is reserved for use by the caller, or it fails, indicating that it is currently in use by other software.

Using an interrupt here means that an interrupt handler is provided, together with an interrupt level, an enabling flag, and whether the interrupt is edge- or level triggered. New values for these instance setup parameters can be set using function **intInstance-Setup**, while the ones currently in use can be obtained using function **intGetInstance-Setup**. The handler for a particular interrupt may be any C function with prototype according to type intHandler (see type definitions), or Null. Interrupt handlers must contain a handler pragma either **TCS_handler** or **TCS_interruptible_handler** in their function bodies. These pragmas cause the generation of specific function prologue and epilogue code which are required for interrupt handlers. The difference between TCS_handler and TCS_interruptible_handler is that TCS_handler clears PCSW.IEN at the beginning of the handler, to be restored at the end, and hence runs with interrupts *disabled*. A **TCS_handler** is fully equivalent to a **TCS_interruptible_handler** which starts with a call to intClearIEN and ends with a call to **intSetIEN**.

In case the current handler for a particular interrupt is not equal to Null, the following determines whether the corresponding interrupt instance is enabled in MMIO.IMASK:

- the setup parameter enabled
- the setup parameter priority
- the global interrupt priority value (to be set using intSetup).

The interrupt will be enabled in MMIO.IMASK if enabled is equal to True, and if the interrupt's priority is larger than, or equal to the global interrupt priority value.

By this, tmInterrupts emulates an interrupt priority, or interrupt level mechanism in which a global interrupt priority, or interrupt level can be set at any time (using function intSetPriority), which causes all interrupts with a lower level/priority to be masked in the IMASK. Interrupts can be assigned a new priority at any time using function intInstanceSetup, and the global priority value can be changed at any time using function intSetup; this will automatically cause a corresponding, proper update of the IMASK.

If the interrupt is enabled in MMIO.IMASK (and if the interrupts are globally enabled, see further), then occurrence of the interrupt causes the handler to be invoked. If not, then occurred interrupts are kept pending until they get enabled. Note that the interrupt pending condition (MMIO.IPENDING) is automatically cleared by the hardware *only* for edge-triggered interrupts; level triggered interrupts must be cleared by means of an explicit acknowledge of the corresponding device.

The handler may also be Null. In this case the enabled flag is overruled, and the corresponding interrupt is not enabled. This can be used for keeping a particular interrupt reserved without actually installing a handler.

In contrast to **intInstanceSetup**, which set instance specific properties, functions intSetup sets properties which affect all interrupts: the global interrupt enable bit (IEN), and the global interrupt priority level.

Interrupts can be globally enabled, or disabled using functions **intSetup**, or the shortcuts **intSetIEN**, **intClearIEN**, and **intRestoreIEN**. Actually, these functions interface to the PCSW.IEN bit. **intClearIEN** massively disables all interrupts of priorities intPRIO_0. Interrupts of priority 7 can only be masked by explicitly clearing its enabled flag using intInstanceSetup. The following points should be considered:

1. Since a lot of software assumes that all interrupts can be disabled by intClearIEN (i.e. by disabling the interrupts), the handlers at priority 7 (NMI handlers, or Non Maskable Interrupt handlers) must only be used in exceptional situations, and with a lot of consideration.
2. Disabling the interrupts, by either intClearIEN or in TCS_handlers, should be performed for durations of at most a few tenths of microseconds in order to maintain real time response on time critical devices.
3. Similarly, changing the global priority to a value different from intPRIO_0, should not be performed for a longer amount of time, because this also can disable interrupts.
4. The global interrupt level, and the PCSW.IEN are considered as part of the task context by some real time operating systems (notably pSOS). This means in case of task switching while during a condition of disabled interrupts or changed priority, a new task might become active with a different priority, or enabled interrupts. Changing the priority, or disabling interrupts should hence be performed only during small critical sections in which it is guaranteed that no scheduling takes place. Scheduling can be caused by the following: some pSOS calls (see psos reference manual), even for non-preemptive tasks, even with interrupts disabled; a system time tick, when interrupts are enabled, and when the current task is preemptive. The pSOS functions ienter and ireturn, which are intended for use in interrupt handlers, effectively cause non-preemptiveness for the duration of the handler. These calls must be used in critical interrupt handlers, in order to prevent an involuntary context switch during the handler due to, for instance, a system timer tick. <8bat>u
5. Printing, or allocating system resources (malloc) is generally not a good idea in interrupt handlers.
6. Any interrupt handler operating in level triggered mode must acknowledge the interrupt before it terminates or before it reenables the interrupts. In particular, the handler can never be a TCS_interruptible handler, *unless* it is able to acknowledge the interrupt at its very start without doing function calls.

Examples

The following can be used to install and enable a video in interrupt handler. The handler has prio 4, and hence it raises the global level to its own level during its execution to prevent less important interrupts to come through. The level is restored at the end. Note that this priority changing must be embraced by ienter/ireturn to prevent a timer context switch during the time the priority level is modified. Also, the handler must (initially) be non-interruptible, otherwise it can be interrupted by a lower priority interrupt before it had the opportunity to raise the global interrupt level. When setup in this way, interrupts can be enabled in the part of the handler which is marked with a "**...**" using function **intSetIEN**, without the danger of a lower priority interrupt becoming serviced before the **VINHandler** has completed.

Note that this protocol is only necessary when we want to set the interrupt enable bit (IEN) at some point during execution of the handler (to allow higher priority interrupts to be serviced); otherwise, when this is not an issue, only the **TCS_handler** pragma suffices, and neither the ienter/ireturn nor the intSetPriority is necessary.

At the end of the example, the interrupt is given up using an intClose; this implicitly deinstalls the handler and disables the interrupt.

```
#include "tm1/tmInterrupts.h"
static void VINHandler(){
   #pragma TCS_handler          /* -- IEN automatically cleared */
   intPriority_t saved_prio;
   ienter();
   saved_prio = intSetPriority(intPRIO_4);
   ...
   intSetPriority(saved_prio);
   ireturn();
}
intInstanceSetup_t setup;
setup.enabled         = True;
setup.handler         = VINHandler;
setup.level_triggered = True;
setup.priority        = intPRIO_4;
if( intOpen(intVIDEOIN)                  != 0 ) { error(); }
if( intInstanceSetup( intVIDEOIN, &setup) != 0 ) { error(); }
if( intClose(intVIDEOIN)                 != 0 ) { error(); }
```

The following illustrates how an individual setup parameter e.g. the interrupt priority can be modified.

```
#include "tm1/tmInterrupts.h"
intInstanceSetup_t setup;
if( intGetInstanceSetup( intVIDEOIN, &setup != 0 ) { error();}
setup.priority = intPRIO_2;
if( intInstanceSetup   ( intVIDEOIN, &setup != 0 ) { error();}
```

The following illustrates how interrupts can be temporarily disabled, even when it is not known whether this already was the case:

```
#include "tm1/tmInterrupts.h"
Int ien;
ien= intClearIEN();
{
```



```
    counter++;
}
intRestoreIEN(ien);
```

See also the full example in $(TCS)/examples/peripherals/interrupts

IMPORTANT
Interrupt handlers must be compiled using handler pragma TCS_handler or TCS_interruptible_handler.

TMInterrupts API Data Structures

This section describes the TMInterrupts API data structures.

intInterrupt_t

```
typedef enum {
   intINT_0,  intINT_1,  intINT_2,  intINT_3,
   intINT_4,  intINT_5,  intINT_6,  intINT_7,
   intINT_8,  intINT_9,  intINT_10, intINT_11,
   intINT_12, intINT_13, intINT_14, intINT_15,
   intINT_16, intINT_17, intINT_18, intINT_19,
   intINT_20, intINT_21, intINT_22, intINT_23,
   intINT_24, intINT_25, intINT_26, intINT_27,
   intINT_28, intINT_29, intINT_30, intINT_31,

   intTRI_USERIRQ    = intINT_4,
   intTIMER1         = intINT_5,
   intTIMER2         = intINT_6,
   intTIMER3         = intINT_7,
   intSYSTIMER       = intINT_8,
   intVIDEOIN        = intINT_9,
   intVIDEOOUT       = intINT_10,
   intAUDIOIN        = intINT_11,
   intAUDIOOUT       = intINT_12,
   intICP            = intINT_13,
   intVLD            = intINT_14,
   intV34            = intINT_15,
   intPCI            = intINT_16,
   intIIC            = intINT_17,
   intJTAG           = intINT_18,
   intTHERM          = intINT_20,
   intHOSTCOMM       = intINT_28,
   intAPP            = intINT_29,
   intDEBUGGER       = intINT_30,
   intRTOS           = intINT_31
} intInterrupt_t;
```

Description

These are the possible interrupt sources. You can use the numeric, or the symbolic names. This enum is used as a parameter for the following functions:

```
intInstanceSetup      intOpen    intClear        intRaise
intGetInstanceSetup   intClose   intGetPending   intRaise_M
```

intPriority_t

```
typedef enum {
   intPRIO_0,
   intPRIO_1,
   intPRIO_2,
   intPRIO_3,
   intPRIO_4,
   intPRIO_5,
   intPRIO_6,
   intPRIO_NMI
} intPriority_t;
```

Description

These are the legal values for interrupt priority. This enum is used as a field by the structures **intSetup_t** and **intInstanceSetup_t**, and as a parameter by the function **intSetPriority**.

intCapabilities_t

```
typedef struct {
   tmVersion_t   version;
   Int           numCurrentInstances;
} intCapabilities_t, *pintCapabilities_t;
```

Fields

`version`	This contains the version information for this library. Backward and forward compatibilities can be maintained if software inspects this version and responds accordingly.
`numSupportedInstances`	The number of instances supported. This value is the same as the number of interrupts.
`numCurrentInstances`	The number of instances (interrupts) currently open.

Description

This structure is used as a parameter by the function **intGetCapabilities**.

intSetup_t

```
typedef struct{
   Bool            enabled;
   intPriority_t   priority;
} intSetup_t, *pintSetup_t;
```

Fields

`enabled`	Global interrupt enable flag.
`priority`	Global interrupt level.

Description

This structure is used as a parameter by the functions **intSetup** and **intGetSetup**.

intInstanceSetup_t

```
typedef struct {
   Bool            enabled;
   intHandler_t    handler;
   intPriority_t   priority;
   Bool            level_triggered;
} intInstanceSetup_t, *pintInstanceSetup_t;
```

Fields

`enabled`	This field defines if the interrupt should be enabled in IMASK (See comments in header for meaning when **handler == Null**).
`handler`	Installed handler, or null.
`priority`	Handler priority.
`level_triggered`	Triggering mode.

Description

This structure is used as a parameter for the functions **intInstanceSetup** and **intGetInstanceSetup**.

TMInterrupts API Functions

This section presents the TMInterrupts API functions.

intGetCapabilities

```
tmLibdevErr_t intGetCapabilities(
   pintCapabilities_t   *cap
);
```

Parameters

`cap`	Pointer to a pointer of type **intCapabilities_t**.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts, in the debug version of the device library, when **cap** is null.

Description

The function retrieves global capabilities from a static variable (for internal use only).

intSetup

```
tmLibdevErr_t intSetup(
   intSetup_t   *setup
);
```

Parameters

`setup`	Pointer to a struct containing setup data.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted by the debug version of the device library if the setup parameter is null.
`INT_ERR_STRUCT_CHANGED`	Asserted by the debug version of the device library if the **intInstanceSetup_t** data structure has been modified in the header file without a corresponding change in the library source.

Description

This function is used to set or to change global parameters. It modifies the current priority, and modifies the IEN flag according to **setup–>enabled**.

intGetSetup

```
tmLibdevErr_t intGetSetup(
   intSetup_t   *setup
);
```

Parameters

`setup`	Pointer to structure where the data is to placed.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted by the debug version of the device library if the setup parameter is null.
`INT_ERR_STRUCT_CHANGED`	Asserted by the debug version of the device library if the **intInstanceSetup_t** data structure has been modified in the header file without a corresponding change in the library source.

Description

This function retrieves global parameters. It disables the interrupts, gets the value of IEN, the current priority, and restores interrupts.

intInstanceSetup

```
tmLibdevErr_t intInstanceSetup(
   intInterrupt_t       instance,
   intInstanceSetup_t  *setup
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	The specified interrupt instance has not been opened. It is asserted by the debug version of the device library if the specified instance has not been opened.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted by the debug version of the device library if the setup parameter is NULL.
`INT_ERR_STRUCT_CHANGED`	Asserted by the debug version of the device library if the **intInstanceSetup_t** data structure has been modified in the header file without a corresponding change in the library source.

Description

This function is used to set or change instance parameters.

intGetInstanceSetup

```
tmLibdevErr_t intGetInstanceSetup(
   intInterrupt_t       instance,
   intInstanceSetup_t  *setup
);
```

Parameters

`instance`	Instance value.
`setup`	Pointer to structure in which to place the data.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted by the debug version of the device library if the specified instance has not been opened.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted by the debug version of the device library if the setup parameter is null.
`INT_ERR_STRUCT_CHANGED`	Asserted by the debug version of the device library if the **intInstanceSetup_t** data structure has been modified in the header file without a corresponding change in the library source.

Description

This function retrieves instance parameters.

intOpen

```
tmLibdevErr_t intOpen(
   intInterrupt_t   interrupt
);
```

Parameters

`interrupt`	Interrupt number.

Return Codes

`TMLIBDEV_OK`	Success.
`INT_ERR_ALREADY_OPEN`	Returned if the interrupt is currently open.

Description

This function reserves the use of the specified interrupt.

intClose

```
tmLibdevErr_t intClose(
   intInterrupt_t   instance
);
```

Parameters

`instance`	Interrupt number.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted by the debug version of the device library if the specified instance has not been opened.

Description

This function deallocates the interrupt instance and uninstalls its handler when it has one. It disables the interrupt if it is enabled.

intSetPriority

```
intPriority_t intSetPriority(
   intPriority_t   priority
);
```

Parameters

`priority`	Interrupt priority.

Return

The function returns the previous global priority. This function always completes successfully.

Description

The function sets the global interrupt priority, or interrupt level. The same result can also be achieved using the function **intSetup**.

The function returns the previous global priority.

intSetIEN

```
UInt intSetIEN( void );
```

Parameters

None.

Return

The function returns the previous PCSW. The previous setting of the global interrupt enabling bit can be extracted from this. The return value can also be passed unmodified to the **intRestoreIEN** function to restore the global interrupt enabling bit.

The function always completes successfully.

Description

This function sets the global enabling bit and returns the previous PCSW. This function gets the state of the PCSW, and writes IEN to the IEN flag of the PCSW.

You can also use the **intSET_IEN** macro in place of the **intSetIEN** function. The advantage is less overhead. Following is a definition of the **intSET_IEN** macro:

```
/* Note: intSET_IEN and intCLEAR_IEN can only be used in the form
 *       "x = intSET_IEN();" */
#define intIEN        0x400
#define intSET_IEN()  ((readpcsw()&intIEN)!=0); writepcsw(intIEN,intIEN);
```


intClearIEN

```
UInt intClearIEN( void );
```

Parameters

None.

Return Codes

The function returns the previous PCSW. The previous setting of the global interrupt enabling bit can be extracted from this. The return value can also be passed unmodified to the **intRestoreIEN** function to restore the global interrupt enabling bit to its previous state.

Description

This function clears the global enabling bit and returns the previous PCSW. This function gets the state of the PCSW, and writes 0 to the IEN flag of the PCSW.

You can also use the **intCLEAR_IEN** macro in place of the **intClearIEN** function. The advantage is less overhead. Following is a definition of the **intCLEAR_IEN** macro:

```
/* Note: intSET_IEN and intCLEAR_IEN can only be used in the form of
         "x = intSET_IEN();" */
#define  intIEN          0x400
#define  intCLEAR_IEN()  ((readpcsw()&intIEN)!=0); writepcsw(0,intIEN);
```

intRestoreIEN

```
void intRestoreIEN(
   UInt   ien
);
```

Parameters

`ien`	Interrupt enable information.

Return

Nothing is returned.

Description

This function restores the specified interrupt enabling, which might have been returned by a previous call to **intSetIEN** or **intClearIEN**. It writes the IEN flag from the PCSW according to the value given by the caller.

You can also use the **intRESTORE_IEN** macro in place of the **intRestoreIEN** function. The advantage is less overhead. Following is a definition of the **intRESTORE_IEN** macro:

```
/* Note: intSET_IEN and intCLEAR_IEN can only be used in the form
         "x = intSET_IEN();" */
#define  intIEN               0x400
#define  intRESTORE_IEN(ien)  writepcsw(mux(ien,intIEN,0),intIEN);
```


intClear

```
tmLibdevErr_t intClear(
   intInterrupt_t  instance
);
```

Parameters

`instance`	Interrupt number.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted by the debug version of the device library if the specified instance has not been opened.

Description

This function clears an interrupt that may or may not be pending. This calls the macro **intAckCLEAR**, and it is compiled with **#pragma TCS_atomic**. It was used to circumvent hardware bug 21388 present in TM-1000 (but not in TM-1100).

intRaise

```
tmLibdevErr_t intRaise(
   intInterrupt_t  instance
);
```

Parameters

`instance`	Interrupt number.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted by the debug version of the device library if the specified instance has not been opened.

Description

This function causes a software interrupt to be raised on the current node. This calls the macro **intAckPending**.

intGetPending

```
tmLibdevErr_t intGetPending(
   intInterrupt_t   instance,
   Bool   *pending
);
```

Parameters

`instance`	Interrupt number.
`pending`	Pointer in which to return the pending status.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted by the debug version of the device library if the specified instance has not been opened.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted by the debug version of the device library if a null pending argument is presented.

Description

This function inspects whether the specified interrupt is pending. It sets ***pending** to 1 if an interrupt is pending, according to the macro **intCheckPENDING**, and 0 otherwise.

intRaise_M

```
tmLibdevErr_t intRaise_M(
   UInt             node_number,
   intInterrupt_t   instance
);
```

Parameters

`node_number`	Number of node whose instance to raise.
`instance`	Interrupt number.

Return Codes

`TMLIBDEV_OK`	Always returned.
`INT_ERR_INVALID_NODE`	Asserted by the debug version of the device library if the argument node number is greater than or equal to the number of nodes.

Description

This function causes a software interrupt to be raised on the current (or another) node.

Chapter 22

TMIntPins API

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

PCI Interrupt Pins API Overview

The TriMedia processor includes four pins called intA through intD. IntA is generally assigned to be the interrupt allocated to the PCI bus.TriMedia's PCI configuration registers specify this. The other three pins can be used as general purpose I/O. Figure 10-9 in the TriMedia data book shows the logic used for these pins in schematic form. These pins are open drain and must be pulled up for proper operation. Please note that early TM IREF boards incorrectly connected all of these pins to the PCI bus, leading to dangerous results.

The PCI interrupt pins are modeled according to the general TriMedia device model, that is, each interrupt (for example, **pinPin_A**, **pinPin_D**) is considered an instance which must be opened (by calling the **pinOpen** function) before it can be used, and must be closed (by calling the **pinClose** function) when it to be used no longer, to make it available for other software components. If the request to open succeeds, it means that the requested interrupt pin is reserved for use by the caller. If it fails, it means that it is currently in use by other software.

Using an interrupt pin means that the pin is used for reading values from peripherals connected to the pin, or for writing values to such a peripheral. Optionally, an interrupt handler with priority can be specified using function **pinInstanceSetup**. When no interrupt handler is required, a null handler can be provided. A non-null handler will be automatically installed using the tmInterrupt library. Conversely, associated interrupt handlers will be automatically uninstalled for interrupt pins when the pins are closed. The **pinInstanceSetup** function can be repeatedly used for modifying one or more of the pin's set up fields. When only a few of the fields are to be changed, usually a call to the **pinGetInstanceSetup** function is necessary to get the current value of the other fields. After the pin has been set up, the action functions for getting or setting the value can be called for operating the pin.

IMPORTANT:
Interrupt handlers must be compiled using handler pragma TCS_handler or TCS_interruptible_handler. Read also the notes in tmInterrupts.h.

NOTE:
Any interrupt handler associated with an interrupt pin operates in level triggered mode. This means that the handler must acknowledge the interrupt before it terminates or before it enables the interrupts. In particular, the handler can never be a TCS_interruptible handler, UNLESS it is able to acknowledge the interrupt at its very start without doing function calls.

EXAMPLE

The following code can be used to output a value to an interrupt pin, or to read from it. In the second part, it is set up to trigger an interrupt on receiving a FALSE to TRUE transition on the pin. Such a transition can be made either in software, using **pinSet**, or by an actual voltage transition on the pin itself. However, when expecting real pin signals,

the software setting of the pin should be FALSE (which is the initial value after **pinOpen**), otherwise this software setting will mask the hardware setting. In general, it is a good idea to leave **open_collector** to True, and use a pullup device when the pin signal is used as output.

```
#include “tm1/tmIntPins.h”
static void Handler(){
   #pragma TCS_handler
   ....
}
Bool                 value;
pinInstanceSetup_t   setup;

setup.open_collector = True;
setup.handler        = Null;
setup.priority       = intPRIO_0;

if( pinOpen(pinPin_B)                != 0  ) {error();}
if( pinInstanceSetup(pinPin_B, &setup!= 0) ) {error();}

pinSet(pinPin_B, False);
pinSet(pinPin_B, True );
pinGet(pinPin_B, &value);
...

if( pinGetInstanceSetup(pinPin_B, &setup!= 0) ) {error();}

setup.handler = Handler;
```

See also the full example in $(TCS)/examples/peripherals/intpins.

PCI Interrupt Pins API Data Structures

This section presents the PCI Interrupt Pins device library API data structures.

pinInterruptPin_t

```
typedef enum {
   pinPin_A,
   pinPin_B,
   pinPin_C,
   pinPin_D
} pinInterruptPin_t;
```

Description

This enum defines the interrupt selection. It is used as a parameter type in the following functions:

```
pinInstanceSetup      pinOpen    pinGet   pinGetInstanceCapabilities
pinGetInstanceSetup   pinClose   pinSet
```

pinCapabilities_t

```
typedef struct {
   tmVersion_t   version;
   Int           numSupportedInstances;
   Int           numCurrentInstances;
} pinCapabilities_t, *ppinCapabilities_t;
```

Fields

`version`	version allows the application software to identify and support possible changes to the interface.
`numSupportedInstance`	Number of supported instances, currently four, one for each intPin.
`numCurrentInstances`	Number of instances open, incremented when a new instance is allocated, decremented as each is closed.

Description

This structure is used in the function **pinGetCapabilities.**

pinInstanceCapabilities_t

```
typedef struct {
   intInterrupt_t   interrupt;
} pinInstanceCapabilities_t, *ppinInstanceCapabilities_t;
```

Fields

`interrupt`	The ID of the interrupt used by this pin (that is, intA gets zero, as per table 3-9 in the TriMedia databook).

Description

This structure is used by the function **pinGetInstanceCapabilities**.

pinInstanceSetup_t

```
typedef struct {
   Bool            open_collector;
   intHandler_t    handler;
   intPriority_t   priority;
} pinInstanceSetup_t, *ppinInstanceSetup_t;
```

Fields

`open_collector`	If **open_collector** is true, the IE bit corresponding to this int pin is enabled. See table 10-18 in the TriMedia databook.
`handler`	The handler is an interrupt service routine.
`priority`	Priority for the (optional) interrupt handler.

Description

This structure is used as a parameter by the function **pinInstanceSetup**.

PCI Interrupt Pins API Functions

This section presents the PCI Interrupt Pins API functions.

pinGetCapabilities

```
tmLibdevErr_t pinGetCapabilities(
   ppinCapabilities_t   *cap
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`BOARD_ERR_NULL_FUNCTION`	In the debug version of the library, this assertion is triggered if **cap** is null.

Description

Provided so that a system resource controller can find out about the SSI library before installing it, and it fills in the address of a static capabilities structure. The **cap** pointer is valid until the pin library is unloaded.

pinGetInstanceCapabilities

```
tmLibdevErr_t pinGetInstanceCapabilities(
   pinInterruptPin_t           instance,
   pinInstanceCapabilities_t  *cap
);
```

Parameters

instance	Instance value.
`cap`	Pointer to struct of type **pinInstanceCapabilities_t** where data is to be placed.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if the instance does not match the owner.
`BOARD_ERR_NULL_FUNCTION`	In the debug version of the library, this assertion is triggered if **cap** is null.
`PIN_ERR_STRUCT_CHANGED`	In the debug version of the library, this assertion is triggered if the size of the struct, **pinInstance-Capabilities_t** has changed.

Description

This function is used to retrieve instance capabilities.

pinInstanceSetup

```
tmLibdevErr_t pinInstanceSetup(
   pinInterruptPin_t    instance,
   pinInstanceSetup_t  *setup
);
```

Parameters

`instance`	Instance value.
`setup`	Pointer to struct of type **pinInstanceSetup_t** where data is placed by caller.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if the instance does not match the owner.
`BOARD_ERR_NULL_FUNCTION`	In the debug version of the library, this assertion is triggered if **setup** is null.

Description

This function is used to set up an interrupt pin. The setup structure specifies input/output mode, and an optional interrupt handler.

pinGetInstanceSetup

```
tmLibdevErr_t pinGetInstanceSetup(
   pinInterruptPin_t    instance,
   pinInstanceSetup_t  *setup
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if the instance does not match the owner.
`BOARD_ERR_NULL_FUNCTION`	In the debug version of the library, this assertion is triggered if **setup** is null.

Description

The function disables interrupts, retrieves the data from the static variable to ***setup**, then restores interrupts.

pinOpen

```
tmLibdevErr_t pinOpen(
   pinInterruptPin_t   pin
);
```

Parameters

`pin`	Pin to be opened.

Return Codes

`TMLIBDEV_OK`	Success.
`PIN_ERR_ALREADY_OPEN`	The pin has already been opened.
`PIN_ERR_ISR_INSTALLATION_FAILED`	Returned if ISR installation failed.

Description

This function reserves the use of a specified interrupt pin.

pinClose

```
tmLibdevErr_t pinClose(
   pinInterruptPin_t  instance
);
```

Parameters

`instance`	Instance value.

Return Codes

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

Description

This function deallocates the pin instance and uninstall its handler when it has one.

pinGet

```
tmLibdevErr_t pinGet(
   pinInterruptPin_t   instance,
   Bool               *value
);
```

Parameters

`instance`	Instance value.
`value`	Pointer to a variable in which to return the pin's state.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`BOARD_ERR_NULL_FUNCTION`	In the debug version of the library, this assertion is triggered if **value** is null.
`TMLIBDEV_ERR_NOT_OWNER`	In the debug version of the library, this assertion is triggered if the instance does not match the owner.

Description

According to the result of the macro **punchiest**, return 1 or 0.

pinSet

```
tmLibdevErr_t pinSet(
   pinInterruptPin_t  instance,
   Bool               value
);
```

Parameters

`instance`	Instance value.
`value`	New value for pin (low if False).

Return Codes

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

Description

According to **value**, the function will call **pinEnableINT** or **pinDisableINT**.

Chapter 23

TMProcessor API

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

Overview

The tmProcessor API is not in line with the other device library APIs. It can only be used to get general information about the current TM processor the program is running on.The only function, **procGetCapabilities**, returns the following information about the TM processor on which the current program is running.

- The version of the processor in the TriMedia architecture family.
- The revision of the processor.
- The clock frequency on which the processor operates.
- The assigned node number for this processor (to be used in case this processor is part of a multi-processor system).
- The total number of TM processors in this system.
- The type of the host.

TMProcessor API Data Structures

This section presents the TMProcessor API data structures.

procDevice_t

```
typedef enum {
   PROC_DEVICE_UNKNOWN  = 0,
   PROC_DEVICE_TM1000   = 1,
   PROC_DEVICE_TM1100   = 2
} procDevice_t;
```

Fields

PROC_DEVICE_UNKNOWN	Used in case the exact device is not known.
PROC_DEVICE_TM1000	The first member of the TriMedia family.
PROC_DEVICE_TM1100	An extended version of the TM-1000, with an extended Video Out unit, support for XIO, and much more. See the Databook for details.

Description

This type is an enumeration type with values for the different members of the family of TriMedia processors. In general, different family members can have different instruction sets and different sets of MMIO registers.

procRevision_t

```
typedef enum {
   PROC_REVISION_UNKNOWN = 0,
   PROC_REVISION_CTC     = 1,
   PROC_REVISION_1_0     = 2,
   PROC_REVISION_1_1     = 3,
   PROC_REVISION_1_0S    = 4,
   PROC_REVISION_1_1S    = 5,
   PROC_REVISION_1_2     = 6,
   PROC_REVISION_1_3     = 7
} procRevision_t;
```

Fields

PROC_REVISION_UNKNOWN	Used when the revision is not known.
PROC_REVISION_CTC	CPU Test Chip, all versions. These chips are not supported by the libraries and tools.
PROC_REVISION_1_xx	Different revisions of a processor. The 'S' stands for shrink version.

Description

This enumeration type is available to hold the exact revision of the chip. Although different revisions are compatible, small differences may exist that influence the performance of the processor.The chronological order of tapeout of the revisions is the same as the order of the corresponding enumeration fields.

procCapabilities_t

```
typedef struct{
   tmVersion_t      version;
   procDevice_t     deviceID;
   procRevision_t   revisionID;
   UInt             cpuClockFrequency;
   UInt32           nodeNumber;
   UInt32           numberOfNodes;
   tmHostType_t     hostID;
} procCapabilities_t, *pprocCapabilities_t;
```

Fields

`version`	Version of this library component.
`deviceID`	The member of the TriMedia family the program is running on.
`revisionID`	Holds information about the exact revision of the device. It may be needed to work around hardware bugs in earlier versions of a chip.
`cpuClockFrequency`	Clock frequency of the processor [Hz].
`nodeNumber`	In case of a system with multiple TM's, a unique identification for the several processors is provided with the field nodeNumber. It can be used in communication with other TM processors as well as with a possible host processor. For more information on the nodeNumber, refer to tmld Options in Chapter 11 of Book 4, *Software Tools*, Part B.
`numberOfNodes`	Holds the number of TriMedia processors in the current system.
`hostID`	Gives information about the possible host processor. Value tmNoHost indicates a stand alone system. tmTmSimHost indicates this program is run on a simulator **tm(t)sim**. **tmWin32Host** and tmMacHost are used when the the TriMedia is a co-processor in a Win32 or MacOS system respectively. Finally, **tmInvalidHost** is used when the host is unknown.

Description

Holds all information about the current TM processor. A pointer to a global instance of this type is set by the function procGetCapabilities. Next to a field version of type **tmVersion_t** that tells the user the exact version of this library component.

TMProcessor API Functions

This section presents the TMProcessor API functions. There is only one.

procGetCapabilities

```
tmLibdevErr_t procGetCapabilities(
   pprocCapabilities_t   *cap
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Always returned.

Description

The function gets the capabilities of this processor. The function has no discernable side-effects.

Chapter 24

Semaphore API

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

Semaphore API Overview

The TriMedia processor includes a semaphore device which is designed to facilitate interprocessor communication. This is described in chapter 18 of the TriMedia data book(s). This library provides an interface to that semaphore.

Semaphore provides the synchronization functions, **smdevGet** and **smdevRelease**. These functions identify their semaphores using a node number, thereby implicitly knowing that each node has only one semaphore device. The functions solve the following problems:

- They construct a unique 12-bit ID from the current node number (assigned by the downloader).
- They take care of endian swapping at semaphore access in case the BUI_CTL.SE is not equal to PCSW.BSX (in other words: if the current TM-1 endianness differs from the endianness from the host).
- They implement the proper spinning protocol described in the data book.

The TriMedia semaphore devices currently do NOT conform to the TriMedia device model. The reason for this is that, contrary to other devices, the semaphores are intended for being used across different nodes in a multi-TM-1 system; that is, a particular semaphore on one particular node is allocated for a particular kind of synchronization, and each node which wants to synchronize accesses this semaphore over the PCI.

Example

The following can be used to protect incrementing a counter variable in shared memory which is used by multiple TM1's in a multiprocessor system:

```
#define THE_SEMDEV 0
UInt  *shared_counter;
semdevGet(THE_SEMDEV,True);
{
   (*shared_counter)++;
}
semdevRelease(THE_SEMDEV);
```

Semaphore API Functions

This section presents the Semaphore API device library functions.

semdevGet

```
tmLibdevErr_t semdevGet(
   UInt   node_number,
   Bool   wait
);
```

Parameters

`node_number`	Node of semaphore to claim.
`wait`	When **wait** is True, the function will "busy wait" until the semaphore can be acquired. Otherwise the function makes only one attempt and **SEM_ERR_NOT_ACQUIRED** is returned if the attempt failed.

Return Codes

`TMLIBDEV_OK`	Success.
`SEM_ERR_NOT_ACQUIRED`	See **wait** above.

Description

This function will attempt to acquire the semaphore device on the specified node.

Implementation Notes

See the appropriate TriMedia data book. This function uses the current node number as base for the 12-bit ID.

semdevRelease

```
tmLibdevErr_t semdevRelease(
   UInt   node_number
);
```

Parameters

node_number	Node of semaphore to be freed.

Return Codes

TMLIBDEV_OK	Success.
SEM_ERR_NOT_ACQUIRED	Returned if the node number is invalid.

Description

This function releases the semaphore device on the specified node.

Chapter 25

Timers API

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

Timers API Overview

The TriMedia timers are modeled according to the general TriMedia device model, that is, each of the three timers is considered an "instance" which must be "opened" before it can be used, and must be "closed" when no longer used, to make it available for other software components. Opening either succeeds, which means that an (anonymous) unused timer has been reserved for use by the caller, or it fails, indicating that all timers are currently in use by other software.

"Using" a timer here means that the timer is configured to get its events from a specific source, followed by monitoring the events via the timer's value, either by polling, or by awaiting an associated interrupt. See details in the appropriate TriMedia data book.

Opened timers can be set up to relatively constant settings, which are modulus, prescale, source and optional interrupt handler with interrupt priority. When no interrupt handler is required a null handler can be provided, otherwise the handler will be automatically installed using the tmInterrupt library. Conversely, interrupt handlers will be automatically uninstalled for timers which are closed.

The **timInstanceSetup** function can be repeatedly used to modify one or more of the timer's parameters; any changes will be correctly updated in MMIO. When only a few of the parameters are to be changed, usually a call to the **timGetInstanceSetup** function is necessary to get the current value of the other parameters.

After the timer has been set up, the action functions for getting or setting the value, or for starting/stopping it can be called for "operating" the timer.

IMPORTANT
Interrupt handlers must be compiled using handler pragma TCS_handler or TCS_interruptible_handler. Read also notes in tmInterrupts.h.

Example

Use the following to set up a timer that generates an interrupt every 10 µs.

```
#include "tm1/tmTimers.h"

 static void Handler(){
   #pragma TCS_handler
 ....
}

Int                 timer;
timInstanceSetup_t  setup;
UInt32              cycles;
UInt32              ten_us_in_nano_seconds = 10000;

timToCycles( ten_us_in_nano_seconds, &cycles);

setup.source   = timCLOCK;
setup.prescale = 1;
setup.modulus  = cycles;
setup.handler  = Handler;
```



```
setup.priority = intPRIO_4;
setup.running  = True;

if( timOpen(&timer)                  != 0 ) { error(); }
if( timInstanceSetup( timer, &setup) != 0 ) { error(); }
```

See also the full example in $(TCS)/examples/peripherals/timers.

Timers API Data Structures

This section presents the Timers data structures. They are contained in the file tmTimers.h.

timSource_t

```
typedef enum {
   timCLOCK,
   timPRESCALE,
   timTRI_TIMER_CLK,
   timDATABREAK,
   timINSTBREAK,
   timCACHE1,
   timCACHE2,
   timVI_CLK,
   timVO_CLK,
   timAI_WS,
   timAO_WS,
   timV34_RXFSX,
   timV34_IO2
} timSource_t;
```

Fields

`timCLOCK`	Source Name: CPU clock.
`timPRESCALE`	Source Name: prescaled CPU clock.
`timTRI_TIMER_CLK`	Source Name: external clock pin.
`timDATABREAK`	Source Name: data breakpoints.
`timINSTBREAK`	Source Name: instruction breakpoints.
`timCACHE1`	Source Name: cache event 1.
`timCACHE2`	Source Name: cache event 2.
`timVI_CLK`	Source Name: video in clock pin.
`timVO_CLK`	Source Name: video out clock pin.
`timAI_WS`	Source Name: audio in word strobe pin.
`timAO_W`	Source Name: audio out word strobe pin.
`timV34_RXFSX`	Source Name: V34 receive frame sync pin.
`timV34_IO2`	Source Name: V34 transmit frame sync pin.

Description

This enum differentiates among the various timer sources. It is used in the struct **timInstanceSetup_t**.

timCapabilities_t

```
typedef struct{
   tmVersion_t   version;
   Int           numSupportedInstances;
   Int           numCurrentInstances;
} timCapabilities_t, *ptimCapabilities_t;
```

Fields

`version`	Contains the version information for this library. Backward and forward compatibilities can be maintained if software inspects this version and responds accordingly.
`numSupportedInstances`	The number of instances supported. This value is the same as the number of timers.
`numCurrentInstances`	The number of instances (timers) currently open.

Description

This struct is used as a parameter in the function **timGetCapabilities**.

timInstanceCapabilities_t

```
typedef struct {
   intInterrupt_t   interrupt;
} timInstanceCapabilities_t, *ptimInstanceCapabilities_t;
```

Fields

`interrupt`	Enum which specifies 32 interrupt names, defined in tmInterrupts.h.

Description

This struct is used in the function **timGetInstanceCapabilities**.

timInstanceSetup_t

```
typedef struct{
   timSource_t    source;
   UInt32         prescale;
   UInt32         modulus;
   intHandler_t   handler;
   intPriority_t  priority;
   Bool           running;
} timInstanceSetup_t, *ptimInstanceSetup_t;
```

Fields

`source`	This is the enum which selects timer sources.
`prescale`	Range: 1 to 32,768.
`modulus`	Amount to compare value against.
`handler`	Handler callback function, defined in tmInterrupts.h.
`priority`	Enum which specifies 8 interrupt levels, defined in tmInterrupts.h.
`running`	Timer: running = True, stopped = False.

Description

This struct is used as a parameter in the functions **timInstanceSetup** and **timGetInstanceSetup**.

Timers API Functions

This section presents the Timers API functions.

timGetCapabilities

```
tmLibdevErr_t timGetCapabilities(
   timCapabilities_t   *cap
);
```

Parameters

`cap`	Pointer to a structure in which to return capabilities data.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NULL_PARAMETER`	Is asserted by the debug version of the device library if the capabilities parameter, **cap**, is null.

Description

This function returns global capabilities.

timGetInstanceCapabilities

```
tmLibdevErr_t timGetInstanceCapabilities(
   Int                          instance,
   ptimInstanceCapabilities_t  *cap
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned if this instance (timer) has not been opened. Is asserted by the debug version of the device library if this instance (timer) has not been opened.
`TMLIBDEV_ERR_NULL_PARAMETER`	Is asserted by the debug version of the device library if the capabilities parameter, cap, is NULL.

Description

This function returns the capabilities of the specified instance.

timInstanceSetup

```
tmLibdevErr_t timInstanceSetup(
   Int                instance,
   timInstanceSetup_t  *setup
);
```

Parameters

`instance`	Instance value.
`setup`	Pointer to setup data.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Returned if this instance (timer) has not been opened. Is asserted by the debug version of the device library if this instance (timer) has not been opened.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted by the debug version of the device library if the setup parameter is null.

Description

This function will set or change the instance parameters.

timGetInstanceSetup

```
tmLibdevErr_t timGetInstanceSetup(
   Int                  instance,
   timInstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance value.
`setup`	Pointer to a structure in which to return setup data.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted by the debug version of the device library if this instance (timer) has not been opened.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted by the debug version of the device library if the setup parameter is NULL.

Description

This function returns the instance parameters.

timOpen

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

Parameters

`instance`	Instance value.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NO_MORE_INSTANCES`	Returned if the maximum number of instances (timer) have been opened.
`TMLIBDEV_ERR_NULL_PARAMETER`	Is asserted by the debug version of the device library if the instance pointer argument is null.

Description

This function assigns a unique timer instance for the caller.

timClose

```
tmLibdevErr_t timClose(
   Int   instance
);
```

Parameters

`instance`	Instance value.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted by the debug version of the device library if this instance (timer) has not been opened.

Description

This function deallocates the timer instance and uninstalls its handler when it has one.

timGetTimerValue

```
tmLibdevErr_t timGetTimerValue(
   Int       instance,
   UInt32   *value
);
```

Parameters

`instance`	Instance value.
`value`	Pointer to location to place returned timer value.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NOT_OWNER`	Is asserted by the debug version of the device library if the specified instance (timer) has not been opened.
`TMLIBDEV_ERR_NULL_PARAMETER`	Is asserted by the debug version of the device library if the value parameter is null.

Description

This function retrieves the current timer value.

timSetTimerValue

```
tmLibdevErr_t timSetTimerValue(
   Int      instance,
   UInt32   value
);
```

Parameters

`instance`	Instance value.
`value`	New timer value.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted by the debug version of the device library if the specified instance (timer) has not been opened.

Description

This function sets or changes a timer value.

timStart

```
tmLibdevErr_t timStart(
   Int   instance
);
```

Parameters

instance	Instance value.

Return Codes

TMLIBDEV_OK	Always returned.
TMLIBDEV_ERR_NOT_OWNER	Asserted by the debug version of the device library if the specified instance (timer) has not been opened.

Description

This function starts the timer instance. This function is redundant, since its effects can also be achieved using the function **timInstanceSetup**.

timStop

```
tmLibdevErr_t timStop(
   Int   instance
);
```

Parameters

instance	Instance value.

Return Codes

TMLIBDEV_OK	Always returned.
TMLIBDEV_ERR_NOT_OWNER	Asserted by the debug version of the device library if the specified instance (timer) has not been opened.

Description

This function stops the timer instance. This function is redundant, since its effects can also be achieved using the function **timInstanceSetup**.

timToCycles

```
tmLibdevErr_t timToCycles(
   UInt32   nanoseconds,
   UInt32  *cycles
);
```

Parameters

`nanoseconds`	Nanoseconds quantity.
`cycles`	Pointer to a variable in which to place the result.

Return Codes

`TMLIBDEV_OK`	Success.
`TIM_ERR_OVERFLOW`	Returned if the number of cycles calculated cannot be represented in 64 bits.
`TMLIBDEV_ERR_NULL_PARAMETER`	Is asserted by the debug version of the device library if the cycles pointer argument is null.

Description

This function converts nanoseconds to cycles.

timFromCycles

```
tmLibdevErr_t timFromCycles( UI
   UInt32    cycles,
   UInt32   *nanoseconds
);
```

Parameters

`cycles`	Cycles quantity.
`nanoseconds`	Pointer to a variable in which to place the result.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted by the debug version of the device library if the nanoseconds argument is null.

Description

This function converts cycles to nanoseconds.

Chapter 26

DMA API

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

DMA API Overview

The DMA device library is designed to support users who wish to move large blocks of data across the PCI bus. It provides a set of functions to access the DMA operation of the TriMedia PCI interface, which can operate as an autonomous DMA engine, executing block-transfer operations at maximum PCI bandwidth. The interface supports synchronous or asynchronous transfers. The user can specify a transfer as simple as a single address and size, or more complex transactions can be dispatched. This can be useful to support scatter-gather DMA or the transfer of a video image.

Note
See the appropriate TriMedia data book for details on the DMA transfer mechanism.

Unlike device libraries such as the Audio In/Out API and Video In/Out API, the DMA device library does not operate as an exclusive device driver. More than one task can make a DMA request. All DMA requests are queued and executed by the DMA using a deadline-based priority mechanism.

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

The example applications show how the DMA 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 DMA API should be very helpful.

WARNING
Because of a hardware bug in some engineering samples of the TM1000, DMA is not guaranteed to work on any chip earlier than the TM1000 version S1.1 (TM1s 1.1). This is the standard production release.

Demonstration Programs

The DMA API includes a demonstration program, **dmaTest**, which is located in the peripherals/examples/dma directory. The program demonstrates the use of DMA APIs to perform data transfer between PCI and SDRAM.

DMA API Data Structure Descriptions

This section presents the DMA device library data structs.The DMA API is contained in the TriMedia device library, libdev.a. To use the DMA API, you must include the tmDMA.h file. The library libdev.a will be linked automatically.

dmaFunc_t

```
typedef void (*dmaFunc_t)( pdmaRequest_t );
```

Description

This is for the user to provide callback function addresses. It is used as a field in the struct **dmaRequest_t**.

dmaDirection_t

```
typedef enum {
   dmaPCI_TO_SDRAM
} dmaDirection_t;
```

Description

This enum is used to provide direction data to the struct **dmaDescription_t**.

dmaMode_t

```
typedef enum {
   dmaAsynchronous,
   dmaSynchronous,
   dmaSynchronous_By_Polling
} dmaMode_t;
```

Description

This enum is used by the struct **dmaRequest_t** to specify the DMA mode.

- In **dmaAsynchronous** mode, the dispatch function returns immediately and DMA completion is signaled by a completion function, or by polling a flag.
- In **dmaSynchronous** mode, the calling task is blocked, but in a multitasking environment other tasks may continue. This is implemented using the same appModel framework used by fread and others.
- In **dmaSynchronous_By_Polling** mode, the dmaDispatch function spins in a loop waiting for the transaction to complete.

dmaDescription_t

```
typedef struct dmaDescription_t {
   dmaDirection_t   direction;
   Bool             write_and_invalidate;
   UInt             length;
   UInt             nr_of_transfers;
   Pointer          source;
   Pointer          destination;
   UInt             source_stride;
   UInt             destination_stride;
} dmaDescription_t, *pdmaDescription_t;
```

Fields

`direction`	Data movement direction (DMA Write or Read).
`write_and_invalidate`	Corresponds to the T bit described in section 10.6.15 of the databook. If True, a PCI "write and invalidate" transaction is requested.
`length`	Number of bytes to be transferred (Must be a multiple of 4, maximum of 64M.
`nr_of_transfers`	A single DMA request can consist of a number of transfers. For example, during the transfer of a video screen, each line would be a separate transfer.
`source`	Source address.
`destination`	Destination address.
`source_stride`	Distance between the source addresses. For each transfer, the source address is incremented by this value.
`destination_stride`	Distance between the destination addresses. For each transfer, the destination address is incremented by this value.

Description

This struct is used by the struct **dmaRequest_t** to specify the above data.

dmaRequest_t

```
typedef struct dmaRequest_t{
   dmaFunc_t         slack_function;
   dmaFunc_t         completion_function;
   Pointer           data;
   UInt              nr_of_descriptions;
   dmaMode_t         mode;
   UInt              priority;
   volatile Bool     done;
   Int               nrof_retries;
   Pointer           requester;
   pdmaRequest_t     link;
   volatile UInt     current_description;
   dmaDescription_t  descriptions[1];
} dmaRequest_t;
```

Fields

`slack_function`	This (optional) user-specified function is called in **dmaDispatch**. Designed to allow user controlled processing to happen while the DMA is taking place.
`completion_function`	This (optional) user-specified function called in interrupt context to signal the completion of this DMA request.
`data`	Additional information to be used by slack or completion function.
`nr_of_descriptions`	Total number of **dmaDescription_t** structures which make up this request.
`mode`	Specifies the DMA mode. Refer to **dmaMode_t** on page 168 for more information.
`priority`	Request priority (an integer). See description of **dmaRequest** for more details.
`done`	The DMA handler sets this field to 1 on completion of the transaction. You must clear this field before making a request.
`nrof_retries`	Set during operation of **dmaDispatch**. Incremented on every target or master abort.
`requester`	Scratch data for use by **dmaDispatch** only.
`link`	Scratch data for use by **dmaDispatch** only.
`current_description`	Scratch data for use by **dmaDispatch** only.
`descriptions`	Head of a linked list of **dmaDescription_t** structures that describe the transfer to be performed.

Description

This struct is used by the function **dmaDispatch** to provide DMA data.

dmaCapabilities_t

```
typedef struct {
   tmVersion_t   version;
   Int           numSupportedInstances;
   Int           numCurrentInstances;
} dmaCapabilities_t, *pdmaCapabilities_t;
```

Fields

`version`	Version of the DMA library module.
`numSupportedInstances`	Maximum number of instances supported by the DMA library (currently –1, meaning no limit).
`numCurrentInstances`	Number of instances currently open.

Description

A pointer to this structure is returned by the **dmaGetCapabilities** function.

dmaSetup_t

```
typedef struct{
   intPriority_t   priority;
} dmaSetup_t, *pdmaSetup_t;
```

Fields

`priority`	Interrupt priority to be used for the DMA completion interrupt. Use priority zero unless you understand the interrupt priority mechanism and have a reason to raise the priority.

Description

This structure is used by the functions **dmaSetup** and **dmaGetSetup** to set the DMA completion interrupt priority.

DMA API Function Descriptions

This section presents the DMA library API functions.

dmaGetCapabilities

```
tmLibdevErr_t dmaGetCapabilities(
   pdmaCapabilities_t   *cap
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Always returned.

Description

Provided so that a system resource controller can find out about the DMA library before installing it. In addition, this function fills in the address of a static capabilities structure. The **cap** pointer is valid until the DMA library is unloaded.

dmaSetup

```
tmLibdevErr_t dmaSetup(
   dmaSetup_t   *setup
);
```

Parameters

`setup`	Pointer to a structure holding new parameters.

Return Codes

`TMLIBDEV_OK`	Always returned.

Description

Sets the interrupt priority.

This disables interrupts, get properties of the dma interrupt via **intGetInstanceSetup**, then change the priority with **intInstanceSetup**, then restore interrupts. In debug mode, this function can assert on a null setup pointer. Use is optional.

dmaGetSetup

```
tmLibdevErr_t dmaGetSetup(
   dmaSetup_t   *setup
);
```

Parameters

`setup`	Pointer to a structure in which to return setup data.

Return Codes

`TMLIBDEV_OK`	Always returned.

Description

Determines the global setup which currently contains the interrupt priority. In debug mode, this function can assert on a null setup pointer.

dmaOpen

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

Parameters

`instance`	Pointer (returned) to instance data.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assertion is triggered if setup is null.
`DMA_ERR_ISR_INSTALLATION_FAILED`	Returned if interrupt installation failed.

Description

This function assigns a unique instance for the caller. If no DMA interrupt was previously setup, it opens an interrupt with **intOpen**, and sets it up with **intInstanceSetup**. Then, this function enables DMA interrupts by setting the BIU_CTL register (refer to the PCI chapter in the data book).

In debug mode, this function can assert on a null setup pointer.

dmaClose

```
tmLibdevErr_t dmaClose(
   Int   instance
);
```

Parameters

`instance`	Instance value.

Return Codes

`TMLIBDEV_OK`	Always returned.
`TMLIBDEV_ERR_NOT_OPEN`	In the debug version of the library, this assertion is triggered if **dmaOpen** call has not been made.

Description

This function deallocates the specified instance. If this is the last instance, the interrupt service routine opened by **dmaOpen** is shut down.

dmaDispatch

```
tmLibdevErr_t dmaDispatch(
   Int             instance,
   pdmaRequest_t   dma_request
);
```

Parameters

`instance`	Instance value.
`dma_request`	Pointer to the struct containing data for the dispatch function.

Return Codes

`TMLIBDEV_OK`	Success (always returned).
`TMLIBDEV_ERR_NOT_OPEN`	In the debug version of the library, this assertion is triggered if **dmaOpen** call has not been made.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assertion is triggered if **dma_request** is null.

Description

This function allows users to initiate DMA requests using the TriMedia DMA hardware. It can be used in synchronous or asynchronous modes. If no requests are pending, this call immediately initiates a request at the hardware. If other requests are pending, this request is added to a queue using a deadline priority mechanism.

The DMA request structure may specify a number DMA transfers. Each DMA transfer is described as in Chapter 10 of the TM-1 Data Book: a source and destination address and a direction (only DMA between PCI and SDRAM are possible on the TM-1). It is assumed (and therefore not checked) that the provided addresses and the specified direction are consistent. A DMA description (see type above) describes a number of transfers of the same size, from/to a number of equally spaced source and destination start addresses (strides). The source and destination strides are described by two independent fields, which may be different. A DMA request contains a number of such descriptions which are passed to the DMA handler to be executed in order of increasing index in the descriptions array. Immediately after the descriptions are passed, this function executes the specified callback slack_function (when present), and upon completion the DMA handler sets the "done" field and calls the callback completion function (when present). Because of this, DMA transfers can be requested in either of the following modes:

1. Asynchronous mode (field mode = **dmaAsynchronous**).

   After issuing the request, it is the responsibility of the caller to monitor the "done" field for completion. Both slack function and completion function callbacks may be

specified, and the return value of dmaDispatch is unspecified. A custom synchronization mechanism may be built using the callback completion_ function.

2. Synchronous mode (field mode = **dmaSynchronous** or **dmaSynchronous_By_Polling**).

   dmaDispatch will not return before all requested DMA transfers have completed. In principle, in case of dmaSynchronous and when a multitasking operating system (like pSOS) is running, this is implemented by performing a true context switch. When no such OS is running or when the mode is dmaSynchronous_By_Polling, this synchronization is achieved by polling. Note that both the polling and the context switching is performed implicitly/automatically by this function.

Synchronization is convenient to use, but might waste cycles in scheduling or polling for completion. In order to prevent this, the slack_function callback can be used to do some unrelated user actions (probably setting up the next DMA data), giving the DMA request the opportunity to complete. The DMA transfer might be still in progress after the slack_function callback has completed. This might be an indication that this slack_function callback did not have enough work to do, but it might also be that the current request got stalled due to DMA requests by other tasks in a multitasking environment. In any case, such misses can be monitored using the return value of dmaDispatch.

A too-quick slack_function callback might be enlarged, or context switching might be prevented by waiting for the 'done' field to be raised just before completion, or one might simply live with a context switch (and the attendant loss of time).

A priority can be assigned to each DMA transfer request. DMA processing is performed by repeatedly selecting the DMA description which has the highest priority in the currently available set. This means that an issued request will automatically suspend any currently processing DMA request with a lower priority after completion of its current DMA description (that is, processing of a DMA description will never be suspended).

The priority values can be selected from the entire integer range, and hence priorities can be selected as deadlines, i.e. the priority of each DMA request can then be selected as being the moment in time at which completion of the request is required. The DMA library then will schedule the requests in a closest-deadline-first fashion (since lower values are higher priorities). The timescale can be freely chosen, and when a microsecond scale is sufficient, the ANSI clock function can be used.

Chapter 27

IIC API

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

IIC API Overview

This chapter describes the interface to the IIC portion of the TriMedia device library. The IIC library allows users to read and write data over the IIC bus by offering a simple high level API. Synchronous and asynchronous transfers are possible. Also, subaddressing is supported in a convenient way.

The IIC library is implemented in line with the other device libraries. Before starting up a transfer with iicDispatch, an instance needs to be obtained with a call to iicOpen; after use the instance is returned with iicClose.

The functions **iicReadReg** and **iicWriteReg** are still available to read or write one byte, but they are no more than wrappers around a sequence **iicOpen**, **iicDispatch** and **iicClose**.

Due to limitations of the hardware, transfers of certain sizes are not possible on some version of the TM-1000 and TM-1100 (see description of iicDispatch). Also, the TM-1000 will spend a lot of time in the IIC ISR when doing a transfer. This may influence the real-time behavior of the application.

Entry Points

- **iicDispatch**: Initiates a possibly asynchronous IIC transfer.
- **iicReadReg**: Synchronously performs a single byte IIC read.
- **iicWriteReg**: Synchronously performs a single byte IIC write.

The libraries work around a number of hardware bugs in the IIC interface. As a result of these, a number of operations are disallowed. These include 2 byte reads, 4n+1 byte reads, and writes of more than 4 bytes. But transactions supported by the library work with complete reliability.

If it is necessary to work around these limitations, several approaches are available. One is to use the TM-1100 when these limitations are removed by hardware updates. A second is to use the software to write directly to the IIC clock and data pins, thereby executing a “bit-bang” implementation of the IIC protocol.

Demonstration Program

The TriMedia software development kit includes the iic test demonstration program, at TCS/examples/peripherals/iictest. This demonstrates the use of the API, and it can be used to read and write arbitrary IIC addresses. The source code for this demonstration program is provided, as is the source for the library. You can easily modify and rebuild the test program. The library source is provided for reference.

Using the IIC API

Two versions of the device library are provided. The debug version is linked to the example as a default. It checks more error conditions, and it allows you to step through code with the debugger.

IIC API Data Structures

This section presents the IIC API data structures.

iicCapabilities_t

```
typedef struct iicCapabilities_t{
   tmVersion_t    version;
   Int32          numSupportedInstances;
   Int32          numCurrentInstances;
   Int32          largestTransfer;
} iicCapabilities_t, *piicCapabilities_t;
```

Fields

`version`	Version of the iic library.
`numSupportedInstances`	Always –1, meaning no limit.
`numCurrentInstances`	Number of instance at any moment in time.
`largestTransfer`	Maximum number of bytes that can be transferred in one transaction.

Description

Used by **iicGetCapabilities**.

iicSetup_t

```
typedef struct iicSetup_t {
   intPriority_t   interruptPriority;
} iicSetup_t, *piicSetup_t;
```

Fields

`interruptPriority`	Priority of IIC interrupt. The default is zero.

Description

Used by **iicSetup**. Errors codes can be found in tmLibdevErr.h. Standard AV formats can be found in tmAvFormats.h.

iicDirection_t

```
typedef enum {
   IIC_READ   = 1,
   IIC_WRITE  = 0
} iicDirection_t;
```

Description

Direction of transfer on IIC bus.

iicType_t

```
typedef enum {
   IIC_SIMPLE,
   IIC_SUBADDRESS
} iicType_t;
```

Fields

`IIC_SIMPLE`	Simple datatransfer to/from slave.
`IIC_SIMPLE`	Simple datatransfer to/from subaddress of slave.

Description

There are two types of transfers. Simple transfers place one 8-bit (7 bits + R/W) on the bus, and data transfer follows. Subaddress transfers write an extra 8-bit subaddress to the slave before doing the data transfer. Many IIC peripherals support this.

iicMode_t

```
typedef enum {
   IIC_Asynchronous,
   IIC_Synchronous,
   IIC_Synchronous_By_Polling
} iicMode_t;
```

Fields

`IIC_Asynchronous`	Returns immediately after dispatch.
`IIC_Synchronous`	Returns when done. Allows task switching.
`IIC_Synchronous_By_Polling`	Returns when done. Allows no task switching.

Description

The mode of the data transfer determines what happens when the the DSPCPU is waiting for the IIC data transfer to finish. In the asynchronous case, the DSPCPU will not wait at all. In the synchronous case, the current task will wait until the transfer is finished, but other tasks can continue.

In the polling mode, the DSPCPU polls continuously to check for completion of the transfer. Since the polling mode is implemented with a timeout after 30,000 microseconds, this mode is saver when the slave is not well behaved. On the other hand, for extremely slow slaves that cannot handle a (large) transfer within this timeout, this mode cannot be used.

iicFunc_t

```
typedef void (*iicFunc_t) (
   piicRequest_t;
);
```

Fields

`piicRequest_t`	Passed to the completion function.

Description

iicFunc_t is the type of a completion function as it is passed by the user in the request struct **iicRequest_t**. A pointer to the request struct itself is passed as the only parameter to the completion function.

iicRequest_t

```
typedef struct iicRequest_t {
   UInt16          address;
   UInt8           subaddress;
   UInt8           numRetries;
   UInt8           waitBeforeRetry;
   UInt8          *data;
   iicMode_t       mode;
   iicType_t       type;
   iicFunc_t       completion_function;
   volatile Bool   done;
   UInt32          errorCode;
   UInt32          userdata;
   UInt32          index;
   piicRequest_t   link;
   Pointer         requester;
} iicRequest_t, *piicRequest_t;
```

Fields

`direction`	Read or write.
`byteCount`	Number of bytes to read or write (excluding possible subaddress byte).
`address`	IIC address of slave, last bit ignored.
`subaddress`	IIC address, used when type is **IIC_SUBADDRESS**.
`numRetries`	Number of times the transaction is retried after failure.
`waitBeforeRetry`	Number of microseconds waited before a retry.
`data`	Buffer that contains the bytes that are written to slave, or (in case of a read), buffer that is filled with bytes read from slave.
`mode`	Synchronous, polling, or asynchronous transfer mode.
`type`	Simple or subaddress transaction.
`completion_function`	Function executed after the transaction is completed (this function is called from an ISR, so the restrictions that apply to ISRs also apply to this function).
`done`	Boolean that is set to True when the transaction is completed, especially useful for asynchronous transactions.

`errorCode`	When the transaction is completed successfully, this field is set to **TMLIBDEV_OK**; other values indicate an error condition as defined in tmLibdevErr.h.
`userdata`	Extra field that can be used by the completion function.
`index`	Used internally.
`link`	Used internally.
`requester`	Used internally.

Description

This structure is is the fundamental description of an IIC transaction. It is used by **iicDispatch**.

IIC API Functions

This section describes the TriMedia IIC API functions.

iicGetCapabilities

```
extern tmLibdevErr_t iicGetCapabilities(
   piicCapabilities_t   *pcap
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Returned on success.

Description

Copies the address of the **iicCapabilities_t** structure to describe the capabilities of the IIC device library.

iicOpen

```
extern tmLibdevErr_t iicOpen(
   Int    *instance
);
```

Parameters

`instance`	Pointer (returned) to instance variable.

Return Codes

`TMLIBDEV_OK`	Success.

Description

When called for the first time, this function sets up the IIC device and installs the IIC interrupt service routine.

iicClose

```
extern tmLibdevErr_t iicClose(
   Int   instance
);
```

Parameters

`instance`	Instance variable assigned at **iicOpen**.

Return Codes

`TMLIBDEV_OK`	Returned on success.

Description

An internal reference count is decremented. If this is the last open instance number, the ISR is deinstalled (intClose) and the device is shut down.

iicSetup

```
extern tmLibdevErr_t iicSetup(
   piicSetup_t   setup
);
```

Parameters

`setup`	Pointer to **iicSetup_t** structure.

Return Codes

`TMLIBDEV_OK`	Success.
`IIC_ERR_INIT_REQUIRED`	Returned if open has not been called.
(other)	Various errors are returned if setup fails.

Description

Changes the setup of the IIC device as indicated by the values of the fields of the **iicSetup_t** structure.

iicDispatch

```
extern tmLibdevErr_t iicDispatch(
   Int             instance,
   piicRequest_t   request
);
```

Parameters

`instance`	Instance variable assigned at open.
`request`	Pointer to **iicRequest_t** structure.

Return Codes

`TMLIBDEV_OK`	Returned on success.
`IIC_ERR_BAD_ADDRESS`	Address greater than 255.
`IIC_ERR_BAD_COUNT`	Some transfer sizes not yet supported (see below).
`IIC_ERR_TIMEOUT`	TriMedia IIC internal error.
`IIC_ERR_ADDRESS_NACK`	Address not acknowledged by slave.
`IIC_ERR_DATA_NACK`	Data not acknowledged by slave.

Description

This function initiates an IIC transfer. Because several parameters of the transfer can be set in the **iicRequest_t** struct (especially **numRetries** and **waitBeforeRetry**) the transfer can be tuned to the timing characteristics of the slave. Therefore this function better suited for time critical code than **iicReadReg** and **iicWriteReg**.

In case of an asynchronous transfer, the transfer will be started as soon as all previous requests are dealt with, and the function will return immediately. The **iicRequest_t** struct cannot be reused or freed until the transfer is finished (indicated by the done field being set to True). In case of a transfer in synchronous or polling mode, the function returns as soon as the transfer is completed. Clock stretching is always enabled.

Implementation Notes

Because of limitations of the hardware, it is not always possible to transfer a specific number of bytes.

For TM-1000 1.1s, the following bytecounts will result in a **IIC_ERR_BAD_COUNT**:

Read: 2, 5, 9, 13, . . . 4n+1, . . . 251, 252, 253, . . .

Write: 5, 6, 7, 8, . . .

For TM-1100 1.2, the following bytecounts will result in a **IIC_ERR_BAD_COUNT**:

Write: 5, 9, 13, . . . 4n+1, . . .

iicWriteReg

```
extern tmLibdevErr_t iicWriteReg(
   UInt   address,
   Int    subaddress,
   UInt   value
);
```

Parameters

address	IIC address, 8 bits, low bit is R/W.
subaddress	Subaddress used for offset addressing, –1 when not used.
value	Value (byte) to be written.

Return Codes

TMLIBDEV_OK	Returned on success.
IIC_ERR_BAD_ADDRESS	Bad parameter.
IIC_ERR_TIMEOUT	The transaction could not be completed within 300 µs, probably due to an extremely slow slave.
IIC_ERR_ADDRESS_NACK	Address not acknowledged by slave.
IIC_ERR_DATA_NACK	Data not acknowledged by slave.

Description

Synchronously performs a single byte IIC write. Writes the data at the given **address** and **subaddress**. Setting subaddress to a positive number causes a subaddress cycle to be performed. This function calls **iicOpen**and **iicDispatch**in synchronous mode. The instance opened by **iicOpen** is closed by **iicClose**. It is provided for backward compatibility with the old interface.

Note
Because of a hardware bug in the tm1100's I2C block , consecutive reads from or writes to non-existent I2C addresses may appear to succeed, while they actually do not. To prevent this from happening, you should use the function **iicDispatch** instead of **iicWriteReg** when reading from or writing to a non-existing address. Make sure that the **numRetries** field is set to **0**, and that a existing I2C address is used in the following I2C transaction if the **iicDispatch** call returns a value other than TMLIBDEV_OK. See , following, for a demonstration of the workaround.

iicReadReg

```
extern tmLibdevErr_t iicReadReg(
   UInt    address,
   Int    subaddress,
   UInt   value
);
```

Parameters

`address`	IIC address, 8 bits, low bit is R/W.
`subaddress`	Subaddress used for offset addressing,-1 when not used.
`value`	Address of (byte) to be written.

Return Codes

`TMLIBDEV_OK`	Returned on success.
`IIC_ERR_BAD_ADDRESS`	Bad parameter.
`IIC_ERR_TIMEOUT`	The transaction could not be completed within 300 µs, probably due to an extremely slow slave.
IIC_ERR_ADDRESS_NACK	Address not acknowledged by slave.
IIC_ERR_DATA_NACK	Data not acknowledged by slave.

Description

Synchronously performs a single byte IIC read. Reads the data at the given **address** and **subaddress**. Setting subaddress to a positive number causes a subaddress cycle to be performed. This function calls **iicOpen** and **iicDispatch** in synchronous mode. It is provided for backward compatibility with the old interface.

Note
Because of a hardware bug in the tm1100's I2C block , consecutive reads from or writes to non-existent I2C addresses may appear to succeed, while they actually do not. To prevent this from happening, you should use the function **iicDispatch** instead of **iicReadReg** [**iicWriteReg**] when reading from or writing to a non-existing address. Make sure that the **numRetries** field is set to **0**, and that an existing I2C address is used in the following I2C transaction if the **iicDispatch** call returns a value other than TMLIBDEV_OK. See , following, for a workaround.


```
#include<tm1/tmIIC.h>

#define IIC_EEPROM_ADDRESS  0xA0      /* address of boot eeprom on IREF */
#define IIC_MAX_ADDRESS     0xFF
#define EESIZE 16

Int Instance = 0;
Int Byte0    = -1;

void
read_eeprom(void){
   int i;
   iicRequest_t    req;
   tmLibdevErr_t err;
   UInt8 data[EESIZE] = {8};

   req.direction           = IIC_READ;
   req.byteCount           = EESIZE;
   req.address             = IIC_EEPROM_ADDRESS;
   req.subaddress          = 0;
   req.numRetries          = 5;
   req.waitBeforeRetry     = 100;
   req.data                = data;
   req.mode                = IIC_Synchronous_By_Polling;
   req.type                = IIC_SUBADDRESS;
   req.completion_function = Null;

   err  = iicDispatch(Instance, &req);

   printf("Read from EEPROM:\n");
   printf("err = %08x\n", err);
   for (i = 0; i<EESIZE; i++) {
      printf("%02x ", data[i]);
   }
   printf("\n");

   if (Byte0 == -1) Byte0 = data[0];
}
void
recover(void){
   int i;
   tmLibdevErr_t err;
   UInt data;

   for( i = 0; i<100; i++ ){
      err = iicReadReg(IIC_EEPROM_ADDRESS, 0, &data);
      if( err == TMLIBDEV_OK && data == Byte0 ){
         printf("Recovered after %d times\n", i);
         return;
      }
   }
   printf("Cannot recover\n");
}
Bool
try(UInt address){
   iicRequest_t    req;
   tmLibdevErr_t err;
   UInt8 data;

   req.direction           = IIC_READ;
   req.byteCount           = 1;
   req.address             = address;
```

```
   req.subaddress          = 0;
   req.numRetries          = 0;
   req.waitBeforeRetry     = 100;
   req.data                = &data;
   req.mode                = IIC_Synchronous_By_Polling;
   req.type                = IIC_SIMPLE;
   req.completion_function = Null;

   err  = iicDispatch(Instance, &req);

   printf("address 0x%x\n", address);
   printf("err = %08x\n", err);
   printf("%02x ", data);
   printf("\n");

   if(err != TMLIBDEV_OK) recover();

   read_eeprom();
   return (err == TMLIBDEV_OK);
}
main(){
   int i;
   Bool addresses_ok[IIC_MAX_ADDRESS];
   iicOpen(&Instance);
   read_eeprom();

   for( i=0; i<IIC_MAX_ADDRESS; i+=2 ){
      addresses_ok[i] = try(i);
   }
   iicClose(Instance);

   printf("================================================\n");
   printf("On this board the following addresses are used:\n");
   for (i=0; i<IIC_MAX_ADDRESS; i+=2){
      if (addresses_ok[i]) printf("%02x ", i);
   }
   printf("\n================================================\n");
}
```

Figure 2 Workaround Code

Chapter 28

PCI-External I/O (PCI-XIO) API

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

Introduction

The External I/O (XIO) Bus Interface allows the TM-1100 to communicate with external devices. The XIO Bus provides a 16-bit parallel interface to devices such as flash EPROMs. It is modeled on the 68000 processor bus interface and is compatible with existing chips which have 68000 (68K) compatible I/O interfaces. Programmable operating modes allow these control signals to emulate other bus protocols such as x86, ISA bus, PCMCIA, and IDE controller.

The XIO Bus interface provides the following features:

- Simple 8-bit bus for ROM and flash EPROM I/O with control compatibility for 68K and x86 peripheral devices (x86, ISA, PCMCIA, IEEE1284 EPP and IDE devices).
- 16 Mb address range (24 bits of byte addressing).
- Programmable clock speed: 33 MHz PCI (external) or derived from bus clock (internal).
- PCI transfers of 8-bit data between the CPU and external devices.
- Automatic programmable wait state transfer timing.

XIO Operation

The TM-1100 communicates with XIO Bus devices using MMIO transfers. An address register provides the XIO Bus address for the transfer, and a data register sends data to or receives data from the addressed device. The data register also includes control and status bits to minimize the number of MMIO actions necessary to transfer data to or from the XIO bus. Transfers can be 8-bit or 16-bit. 16-bit transfers can be big or little endian.

The XIO Bus controller provides address register auto increment. When active, the XIO bus address is incremented after each XIO bus transfer, in preparation for the next transfer. After initialization, transfers can be performed by MMIO reads and writes of the data register only. This minimizes the number of MMIO operations required per XIO bus transfer, reducing the transfer overhead for bulk transfers such as block moves from flash EPROM.

The XIO Bus provides Programmable Chip Selects (PCS). The PCS feature allows glueless (i.e., no external “glue” logic required) interfaces to XIO Bus devices. Each PCS provides several useful features:

- Programmable automatic wait state generation for I/O transfer timing. Read and write wait states are separately selectable.
- Selectable 8-bit or 16-bit bus width. The 8-bit bus mode allows glueless connection of 8-bit devices such as flash EPROMs.

- Selectable bus type. This allows the selected device to be a 68K, x86/ISA, PCMCIA, or IDE device. The bus type selected determines the function and timing of the variable control lines when the PCS is active.

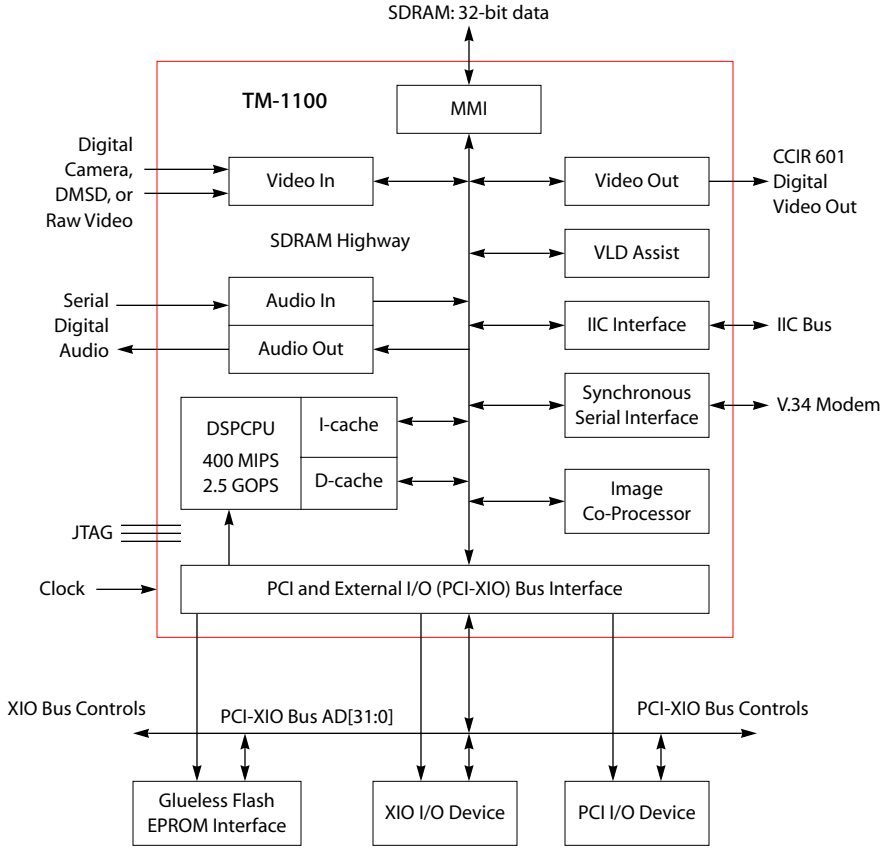

Figure 3 Partial TM-1100 Chip Block Diagram

XIO Example Program

The example program xiotest sets up two buffer areas, one for read and one for write. It compares the XIO output with a reference data to check the correctness of the library. It gives example code on how to set up the XIO initially and provide an ISR to handle the XIO interrupt.

XIO API Data Structures

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

xioCapabilities_t

```
typedef struct {
   tmVersion_t   version;
   Int32         numSupportedInstances;
   Int32         numCurrentInstances;
} xioCapabilities_t, *pxioCapabilities_t;
```

Fields

`version`	Version of this device library.
`numSupportedInstances`	Number of users that can access this simultaneously (currently 1).
`numCurrentInstances`	Number of current users

Description

Used by the function **XIOGetCapabilities**.

xioInstanceSetup_t

```
typedef struct {
   UInt   ct1Address;
   UInt   waitStates;
   UInt   busEnable;
   UInt   internalClockEnable;
   UInt   clockFreqDivider;
} xioInstanceSetup_t, *pxioInstanceSetup;
```

Fields

`ct1Address` — XIO address space.

`waitStates` — The XIO Bus controller has an automatic wait state generator to allow for read and write cycle times of devices on the XIO bus.

`busEnable` — Enable XIO Bus operation. Value: 0 or 1.

`internalClockEnable` — Enable internal clock. When this bit is set, the clock pin becomes an output. Value: 0 or 1.

`clockFreqDivider` — This field defines the clock frequency:

Clock Value	TM-1100 Clocks	PCI-XIO Clock period [ns]	Frequency [MHz]
0	1	10	100.000
1	2	20	50.000
2	3	30	33.333
3	4	40	25.000
...	...	...	...
30	31	310	3.230
31	32	320	3.125

Description

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

XIO API Functions

This section presents the XIO API device library functions.

xioGetCapabilities

```
tmLibdevErr_t xioGetCapabilities(
   pxioCapabilities_t   *cap
);
```

Parameters

`cap`	Pointer to the **xioCapabilities_t** structure.

Return Codes

`TMLIBDEV_OK`	Success (always returned).

Description

Sets the provided pointer to global capabilities.

xioInstanceSetup

```
tmLibdevErr_t xioInstanceSetup(
   Int   Instance,
   xioInstanceSetup_t   *xioSetup
);
```

Parameters

`Instance`	Owner instance.
`xioSetup`	Pointer to **XIOInstanceSetup_t** structure.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Asserts in the debug version if an incorrect instance is passed.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts in the debug version if a **NULL** xioSetup parameter is passed.

Description

This function validates the owner, initializes the registers, resets the XIO and installs the interrupt handler.

xioOpen

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

Parameters

`instance`	Instance pointer.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserted in debug version if a **NULL** instance pointer is passed.
`XIO_ERR_INVALID_PROCESSOR`	Returned if the TriMedia DSPCPU cannot support the XIO interface.

Description

Opens an instance of the XIO device.

xioClose

```
tmLibdevErr_t xioClose(
   Int   instance
);
```

Parameters

`instance`	Device Library instance.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Will assert in the debug version if an incorrect instance is passed.

Description

This function shuts down the device and uninstalls the interrupts.

xioRead Macro

```
xioRead(
   unsigned long   xio_addr,
   unsigned long   value
)
```

Parameters

`xio_addr`	Address from which to read.
`value`	Value at the XIO address.

Return Codes

`TMLIBDEV_OK`	Success.

Description

Read a value from the specified XIO address.

This macro is provided as a convenience. It is defined in *include/tm1/tmXIO.h*.

xioWrite Macro

```
xioWrite(
   unsigned long   xio_addr,
   unsigned long   value
)
```

Parameters

`xio_addr`	Address to which to write.
`value`	Value to write.

Return Codes

`TMLIBDEV_OK`	Success.

Description

Write a value to the specified XIO address.

This macro is provided as a convenience to the user. It is defined in *include/tm1/tmXIO.h*.

Chapter 29

PCI API

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

Overview

This library provides the TriMedia an easy access to the PCI bus. This includes reading, writing, or copying data in 8-, 16-, or 32-bit wide chunks to PCI config space, PCI memory, or I/O space. If you want to have a deeper understanding of the PCI bus, read the PCI bus specification before using this library. This spec can ordered from the PCI Special Interest Group (**http://www.pcisig.com**).

How to Use the PCI Library

Any program that uses the PCI library should include <tm1/tmPCI.h>. The PCI library is contained in libdev.a, and therefore no additional parameter is to be given to **tmcc** during the linking phase. Great care should be taken when using this library, since this library allows a developer to write to or read from any part of the PCI space, and this could lead to system hangs if not used appropriately.

PCI API Functions

This section describes the functions used in the PCI API.

pciAddressFind

```
pciAddressFind (
   unsigned   id,
   unsigned  *CmdStatusAddrPointer
);
```

Parameters

`id`	The id of the device to be found.
`CmdStatusAddrPointer`	Pointer to a buffer that will contain the address of the device in the PCI space.

Return Codes

`TMLIBDEV_OK`	Success.
`PCI_ERR_ADDRESS_FIND`	Unable to find a device with such an ID.

Description

This function tries to find in the list of the devices that are present on the PCI bus, a device that has the id, and returns its address device in the PCI space.

pciConfigRead

```
pciConfigRead (
   UInt32   address,
   UInt32  *data
);
```

Parameters

`address`	Address in the PCI configuration space from which this function should read.
`data`	Pointer to a buffer that will hold the data pointed by **address** on Success.

Return Codes

`TMLIBDEV_OK`	Success.
`PCI_ERR_CONFIG_READ`	The request timed out.

Description

This function reads a 32-bit value from the PCI configuration space and returns it in data.

pciConfigWrite

```
pciConfigWrite (
   UInt32   address,
   UInt32   data
);
```

Parameters

`address`	Address in the PCI configuration space to which this function should write.
`data`	Data to be written at **address**.

Return Codes

`TMLIBDEV_OK`	Success.
`PCI_ERR_CONFIG_WRITE`	The request timed out.

Description

This function writes a 32-bit value in the PCI configuration space at the address specified by **address**.

pciIOReadUInt8

```
pciIOReadUInt8 (
   UInt32   address,
   UInt32  *data
);
```

Parameters

`address`	Address in the PCI I/O space to read from.
`data`	Buffer that will receive the value contained at **address**.

Return Codes

`TMLIBDEV_OK`	Success (always returned).

Description

This function reads a 8-bit value from the PCI I/O space specified by **address**.

pciIOWriteUInt8

```
pciIOWriteUInt8 (
   UInt32   address,
   UInt32   data
);
```

Parameters

`address`	Address in the PCI I/O space to write to.
`data`	Data to be written to **address**.

Return Codes

`TMLIBDEV_OK`	Success (always returned).

Description

This function writes a 8-bit value in the PCI I/O space specified by **address**.

pciMemoryReadUInt32

```
pciMemoryReadUInt32 (
   UInt32   *address,
   UInt32   *data
);
```

Parameters

`address`	Address in the PCI memory to read from.
`data`	Pointer to a buffer that will receive the data read at **address**.

Return Codes

`TMLIBDEV_OK`	Success (always returned).

Description

This function reads a 32-bit value in the PCI memory space specified by **address**.

pciMemoryWriteUInt32

```
pciMemoryWriteUInt32 (
   UInt32  *address,
   UInt32   data
);
```

Parameters

`address`	Address in the PCI memory space to write to
`data`	Data to be written at **address**.

Return Codes

`TMLIBDEV_OK`	Success (always returned).

Description

This function writes a 32-bit value in the PCI memory space at the address specified by **address**.

pciMemoryReadUInt16

```
pciMemoryReadUInt16 (
   UInt16   *data
);
```

Parameters

address	Address in the PCI memory space to read from.
data	Pointer to a buffer that is filled with the value pointed by **address**.

Return Codes

TMLIBDEV_OK	Success (always returned).

Description

This function reads a 16-bit value from the PCI memory space at **address**.

pciMemoryWriteUInt16

```
pciMemoryWriteUInt16 (
   UInt16  *address,
   UInt16   data
);
```

Parameters

address	Address in the PCI memory space to write to.
data	Data to be written at **address**.

Return Codes

TMLIBDEV_OK	Success (always returned).

Description

This function writes a 16-bit value in the PCI memory space at the address specified by **address**.

pciMemoryReadUInt8

```
pciMemoryReadUInt8 (
   UInt8   *address,
   UInt8   *data
);
```

Parameters

`address`	Address in the PCI memory space to read from.
`data`	Pointer to a buffer that is filled with the value pointed by **address**.

Return Codes

`TMLIBDEV_OK`	Success (always returned).

Description

This function reads a 8-bit value from the PCI memory space at **address**.

pciMemoryWriteUInt8

```
pciMemoryWriteUInt8(
   UInt8  *address,
   UInt8   data
);
```

Parameters

`address`	Address in the PCI memory space to which to write.
`data`	Data to be written at **address**.

Return Codes

`TMLIBDEV_OK`	Success (always returned).

Description

This function writes a 8-bit value in the PCI memory space at **address**.

pciMemoryCopy

```
pciMemoryCopy (
   UInt8   *destination,
   UInt8   *source,
   UInt32   length
);
```

Parameters

`destination`	Address to which the data is to be copied.
`source`	Address from which the data is to be copied.
`length`	Number of bytes to be copied.

Return Codes

`TMLIBDEV_OK`	Success.
`PCI_ERR_SDRAM_RANGE`	Invalid destination-source combination.
(other)	Various error codes may be returned from the DMA library (**dmaOpen**, **dmaDispatch**).

Description

This function copies length bytes from **source** to **destination**. One should be in SDRAM, the other should be in the PCI memory space. PCI-to-PCI and SDRAM-to-SDRAM are combinations that are not allowed. SDRAM-to-SDRAM should be performed with a **memcpy**, and a PCI-to-PCI copy should be done by buffering in SDRAM.

The addresses **destination** and **source** must be aligned on a 4-byte boundary.

