Book 9—Communications Support

TriMedia

Version 2.0 beta

Table of Contents

Chapter 1 Synchronous Serial Interface (SSI) API

SSI API Overview 10
Levels of Control 10
SSI Device Library 10
TriMedia BSP 10
SSI MMIO Macro-Only Interface 10
Introduction 10
Notes on the Hardware 11

SSI API Data Structures 11
ssiCapabilities_t 12
ssiInstanceSetup_t 13
ssiFrameSetup_t 14

SSI API Functions 15
ssiGetCapabilities 16
ssiOpen 17
ssiClose 18
ssiInstanceSetup 19
ssiSetFraming 20
ssiGetFraming 21
ssiStop 22
ssiStart 23
ssiConfigure 24
ssiOffHook 25
ssiOnHook 26

Chapter 2 UART API

UART API Overview 28

Uart API Callback Functions 29
tsaUartErrorHandlerFunc_t 30
tsaUartWriteCompletionFunc_t 31
tsaUartReadCompletionFunc_t 31

tsaUartControlHandler_t 32
tsaUartConfigHandler_t 33

Uart API Data Structures 34
tsaUartParity_t 35
tsaUartBaud_t 36
tsaUartConfig_t 37
tsaUartConfigEvent_t 38
tsaUartControl_t 39
tsaUartCapabilities_t 40
tsaUartInstanceSetup_t 41
tsaUartControlMode_t 42

UART API Functions 43
tsaUartGetNumberOfUnits 44
tsaUartGetCapabilities 45
tsaUartOpen 46
tsaUartInstanceSetup 47
tsaUartWrite 48
tsaUartRead 49
tsaUartWriteAbort 50
tsaUartReadAbort 51
tsaUartControl 52
tsaUartInstanceConfig 53
tsaUartClose 54

Chapter 3 V34 Modem API

Overview 56

TriMedia V34 Modem API Overview 57

Interfacing V42 with the Modem 59

Modem Data Structures 60
tmModemCallMode_t 61
tmModemCommandCode_t 62
tmModemCapabilities_t 63
tmModemV25Capabilities_t 64
tmModemStatus_t 65
tmModemStatusCode_t 66
tmModemV42Command_t 67
tmModemV42CommandCode_t 68
tmModemInstanceSetup_t 69

tmModemV8BisConfig_t ... 70
tmModemV8BisStatus_t ... 70
tmModemV8Protocol_t ... 71

Modem Functions ... 72
tmModemAfeInit ... 73
tmModemOpen ... 73
tmModemClose ... 74
tmModemV25SendCommand ... 75
tmModemCommandHandler ... 76
tmModemGetCapabilities ... 76
tmModemInstanceSetup ... 77
tmModemDataWrite ... 78
tmModemStart ... 79
tmModemStop ... 79
tmModemV42Process ... 80
tmModemV42DataRead ... 80
tmModemV42DataWrite ... 81
tmModemV42DataReadPending ... 81
tmModemV42DataWritePending ... 82
tmModemV42CommandHandler ... 82
tmModemV42GetCharFromTxBuffer ... 83
tmModemV42PutCharInTxBuffer ... 84
tmModemV42GetCharFromRxBuffer ... 85
tmModemV42PutCharInRxBuffer ... 86

I/O Handlers ... 87
ModemDataReceiveHandler ... 88
ModemDataTransmitHandler ... 89

Status Handlers ... 90
modemLocalStatusHandler ... 91
V42LocalStatusHandler ... 91

Modem Example: exModem.c ... 92

PPP Example: exModemPPP.c ... 94

Known Bugs ... 95

Chapter 4 1394 FireWire API

IEEE 1394 Overview ... 98
Glossary ... 98

1394 API Overview **99**
Asynchronous Transmission API 101
Isochronous Transmission API 101
Serial Bus Manager API 101
Transaction Layer 102
Memory Utility API 102
API Usage 102
Asynchronous Transmission API 103
Isochronous Transmission API 103
Serial Bus Management API 104
Memory Management API 104

1394 API Enumerated Types **106**
tsa1394FwError_t 107
tsa1394SbmError_t 110
tsa1394SupportMuxMode_t 112
tsa1394Speed_t 113
tsa1394AsynClbkType_t 114
tsa1394ExtTCode_t 115
tsa1394RespCode_t 116
tsa1394trType_t 117
tsa1394trReqStatus_t 118
tsa1394SbEvent_t 119
tsa1394MaxRec_t 120
tsa1394SbmCntrlOp_t 121
tsa1394AddrType_t 122
tsa1394CtrlEvent_t 123

1394 API Data Structures **124**
tsa1394Capabilities_t 126
tsa1394Setup_t 128
tsa1394EUId_t 130
tsa1394DestOffset_t 130
tsa1394FreeRtn_t 131
tsa1394Tdatab_t 132
tsa1394MBlock_t 133
tsa1394MblkBufConfig_t 134
tsa1394DblkLink_t 135
tsa1394BusTime_t 136
tsa1394trDataCnfm_t 137
tsa1394trDataInd_t 138
tsa1394IecTxInfo_t 140

tsa1394IecRxUnpackInfo_t 141
tsa1394IsochHdrInfo_t 142
tsa1394SbResetEventInfo_t 143
tsa1394SbEventInfo_t 144
tsa1394SbmCntrlResetInitParams_t 144
tsa1394SbmCntrlPhyConfigParams_t 145
tsa1394SbmStatusInfo_t 146
tsa1394PhysWriteInfo_t 147
tsa1394CtrlEventInfo_t 147
tsa1394RdBlockReq_t 148
tsa1394RdQuadReq_t 149
tsa1394WrBlockReq_t 150
tsa1394WrQuadReq_t 151
tsa1394RdBlkResp_t 152
tsa1394RdQuadResp_t 153
tsa1394WrResp_t 154
tsa1394LockResp_t 155

1394 API Functions 156

tsa1394GetCapabilities 158
tsa1394Open 158
tsa1394Close 159
tsa1394GetInstanceSetup 160
tsa1394InstanceSetup 161
tsa1394AsyncCreateChannelHandle 162
tsa1394AsyncDestroyChannelHandle 162
tsa1394AsyncRegisterAddressSpace 163
tsa1394AsyncRegisterSbEvent 166
tsa1394AsyncRegisterControlEvent 168
tsa1394AsyncSendRequest 170
tsa1394AsyncSendResponse 174
tsa1394AsyncDeregisterAddressSpace 176
tsa1394AsyncDeregisterSbEvent 177
tsa1394AsyncDeregisterControlEvent 178
tsa1394IsochCreateChannelHandle 179
tsa1394IsochDestroyChannelHandle 180
tsa1394IsochSetupChannel 181
tsa1394IsochStart 182
tsa1394IsochStop 183
tsa1394SbmGetLocalNodeId 184
tsa1394SbmGetSpeed2Node 185
tsa1394SbmGetNodeIdForEuid 186

tsa1394SbmGetMaxRec 187
tsa1394SbmGetBusNodeCount 188
tsa1394SbmCntrlReq 189
tsa1394SbmGetBusId 191
tsa1394MblkInit 192
tsa1394Allocb 193
tsa1394AllocbFromPool 194
tsa1394EsbAlloc 195
tsa1394EsbAllocFromPool 196
tsa1394Freeb 197
tsa1394FreeMsg 198
tsa1394DupB 199
tsa1394DupMsg 200

1394 Callback Functions 201
tsa1394AllocCbFp_t 202
tsa1394FreeCbFp_t 202
tsa1394AddrReqCbFp_t 203
tsa1394AtReqCbFp_t 204
tsa1394SbEventCbFp_t 205
tsa1394CtrlEventCbFp_t 206

Chapter 1

Synchronous Serial Interface (SSI) API

Topic	Page
SSI API Overview	10
SSI API Data Structures	11
SSI API Functions	15

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

SSI API Overview

This module describes the public interface to the TriMedia Synchronous Serial Interface (SSI) portion of the TriMedia device library. For more information on SSI, refer to chapter 16 of the appropriate TriMedia data book.

Levels of Control

The SSI library, like the hardware, was designed with a telecom interface in mind. The SSI library provides control at three levels:

- **High-Level:** SSI Device Library
- **Intermediate-Level:** TriMedia Board Support Package (BSP)
- **Low-Level:** SSI MMIO macro-only interface

SSI Device Library

At the highest level is the SSI device library described in the SSI header file.

TriMedia BSP

At the intermediate level, the SSI device library relies on part of the TriMedia BSP. Using the functions in the board support table, you can customize the functionality of the SSI library.

SSI MMIO Macro-Only Interface

At the lowest level is a macro-only interface, defined by tmSSImmio.h, which is used to set the SSI's MMIO control registers directly. The BSP must use the macro interface exclusively. However, it may be appropriate to use the macro interface directly, particularly in an interrupt service routine.

Introduction

ssiGetCapabilities is provided so that a system resource controller can find out about the SSI library before installing it.

The interface starts by claiming the device, with **ssiOpen**. Currently no sharing can be done, so only one instance can be given out.

Initialization starts with a call to **ssiInstanceSetup**. The SSI device is reset and the Board Support Package (BSP) is queried for any init function. If present, it is called.

The BSP's init function is designed to set any mode controlling MMIO registers. It is not designed to install an ISR, as **ssiInstanceSetup** will install an ISR.

The address of the ISR and its priority are determined with the call to **ssiInstanceSetup**.

ssiStart starts the audio transmit by enabling a transmission on the SSI bus.The interrupt sources which were requested in ssiInstanceSetup are actually enabled here.This is done to prime the transmit fifo before receiving a series of TX interrupts.

ssiSetFraming and **ssiGetFraming** are used to determine the characteristics of a "frame" on the SSI bus. Frames are made up of a number of slots, and some or all of these slots may be valid. For more information, refer to chapter 16 of the databook.

ssiStart and **ssiStop** enable and disable transmission on the SSI bus. The interrupt sources which were requested in ssiInstanceSetup are actually enabled here. This is done so that the user can prime the transmit fifo before receiving a series of TX interrupts.

WARNING
Stopping the SSI bus may necessitate a reset of the off chip peripheral.

In an acknowledgment of the intended use of the SSI as a telecom interface, the entry points, **ssiOffHook** and **ssiOnHook**, are provided with the SSI library. By default, these entry points control the hardware pin IO2. The BSP interface can be used to override this behavior. Refer to tmSSImmio.h for more information.

Notes on the Hardware

The SSI can store 16 words in the transmit FIFO. However, the maximum value indicated by the WAW register is 15 (because it has a 4 bit register field). When the FIFO is empty, WAW has the value 15. When the FIFO is full, WAW has the value 0 and the SSI will ignore any further attempts to add words to the FIFO. Similarly, the receive FIFO can only indicate the storage of 15 words.

Early versions of the TriMedia chip clocked some items like interrupt acknowledge off of the TX clock. In production versions, this clocking was moved to the fast highway domain. Because of this, early silicon is not recommended for development of SSI programs.This API is frozen and will be supported on future chips.

The SSI library depends on the SSI component of the Board Support Package, including dependencies on tmSSImmio.h, tmInterrupts.h, tmLibdevErr.h and MMIO.h

SSI API Data Structures

This section presents the SSI API data structures.

Name	Page
ssiCapabilities_t	12
ssiInstanceSetup_t	13
ssiFrameSetup_t	14

ssiCapabilities_t

```
typedef struct ssiCapabilities_t{
   tmVersion_t   version;
   Int32         numSupportedInstances;
   Int32         numCurrentInstances;
   Char          afeName[16];
   Int32         connectionFlags;
} ssiCapabilities_t, *pssiCapabilities_t;
```

Fields

`version`	Enables version compatibility checking.
`numSupportedInstances`	1.
`numCurrentInstances`	0, until the device is opened, then 1.
`afeName`	Device-specific, for humans to read.
`connectionFlags`	Device-specific, a logic OR of **tmSSIAnalogConnection_t**.

Description

Used by **ssiGetCapabilities**.

ssiInstanceSetup_t

```
typedef struct ssiSetup_t {
   void          (*isr)(void);
   intPriority_t   interruptPriority;
   UInt8           interruptLevelSelect;
   Bool            txInterruptEnable;
   Bool            rxInterruptEnable;
   Bool            changeDetectorInterruptEnable;
   void           *configBuffer;
   UInt32          configBufferLength;
} ssiSetup_t, *pssiSetup_t;
```

Fields

`isr`	Interrupt Service Routine.
`interruptPriority`	SSI needs high priority.
`interruptLevelSelect`	A number between 0 and 15, 10 works well.
`txInterruptEnable`	Interrupt when transmit FIFO is empty.
`rxInterruptEnable`	Interrupt when receive FIFO is full.
`changeDetectorInterruptEnable`	Interrupt when there is a change in state at the IO1 pin. Used for ring detect.
`configBuffer`	Anything else which the BSP might need.
`configBufferLength`	Passed to the BSP's init function.

Description

Used by **ssiInstanceSetup**.

Implementation Notes

interruptLevelSelect sets the mark where interrupts are generated, as the FIFO is 16 deep. Setting **interruptLevelSelect** higher means less interrupts, but consequently, a tighter interrupt latency requirement.

ssiFrameSetup_t

```
typedef struct ssiFrameSetup_t{
   UInt8   validSlotsPerFrame;
   UInt8   slotsPerFrame;
} ssiFrameSetup_t, *pssiFrameSetup_t;
```

Fields

`validSlotsPerFrame`	Referred to as VSS in the data book.
`slotsPerFrame`	Referred to as FSS in the data book.

Description

The data book describes the SSI's transmission of data in frames, each of which have a number of slots, but only some slots are valid. The st7545, for example, uses a frame with 5 slots, but only four of them are valid. The FIFOs are only accessed in valid frames.

SSI API Functions

This section describes the SSI device library functions. Errors are not described in the Return Codes section, but can be found in tmLibDevErr.h. The standard AV formats can be found in tmAvFormats.h

Name	Page
ssiGetCapabilities	16
ssiOpen	17
ssiClose	18
ssiInstanceSetup	19
ssiSetFraming	20
ssiGetFraming	21
ssiConfigure	24
ssiStop	22
ssiStart	23
ssiOffHook	25
ssiOnHook	26

ssiGetCapabilities

```
tmLibdevErr_t ssiGetCapabilities(
   pssiCapabilities_t   *pcap
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.

The function can also return a code produced by the BSP.

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 **pcap** pointer is valid until the SSI library is unloaded.

ssiOpen

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

Parameters

`instance`	Pointer to the (returned) instance.

Return Codes

`TMLIBDEV_OK`	Success.
`IIC_ERR_NO_MORE_INSTANCES`	An instance is already open.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assertion is triggered if the instance is null.

Description

The interface starts by claiming the device, with **ssiOpen**. Currently no sharing can be done, so only one instance can be given out.

The instance variable is assigned. Please remember the variable assigned, as it will be required for further access to the library.

ssiClose

```
tmLibdevErr_t ssiClose(
   Int   instance
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Asserts, in debug mode, on illegal values.

Description

Causes an internal reference count to be decremented. If this is the last open instance, so the ISR is deinstalled, the intINTV34 interrupt is closed with intClose, and the device is shut down with the term_func function from the board API.

Implementation Notes

Errors could also be returned by Int **close** and the BSP's term function.

ssiInstanceSetup

```
tmLibdevErr_t ssiInstanceSetup(
   Int          instance,
   pssiSetup_t  setup
);
```

Parameters

`instance`	The instance, as returned by **ssiOpen**.
`setup`	Pointer to setup structure.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERROR_NOT_OWNER`	Error from **boardGetConfig**.
`TMLIBDEV_ERR_NULL_PARAMETER`	Will assert in debug mode on illegal values.

Description

Used to initialize the SSI and the devices connected to it. It calls the BSP's SSI init function to complete the initialization sequence (look at the Board API). If a isr is provided, it will open and setup the intINTV34 interrupt with **intOpen** and **intInstanceSetup** functions.Leaves the device stopped until **ssiStart** is called.

Implementation Notes

Due to the low latency requirement of the SSI interface, it is recommended that the SSI be run at interrupt priority 6. Priorities lower than 4 may be blocked for as long as a millisecond by host interaction. The **ssiInstanceSetup** function also determines which interrupt sources are enabled, and at which level the FIFO triggers an interrupt.

Responsibility for initialization of an AFE like the Thomson ST7545 used on the IREF board is left to the application. Any initialization data which must be transmitted over the SSI interface must be transmitted by the application. At exit, the interface is stopped. MMIO registers **FSS** (slots per frame), **VSS** (valid slots per frame) are not initialized.

Interrupts are not actually enabled until ssiStart is called.

ssiSetFraming

```
tmLibdevErr_t ssiSetFraming(
   Int               instance,
   pssiFrameSetup_t  frame
);
```

Parameters

`instance`	The instance, as returned by **ssiOpen**.
`frame`	Pointer to frame descriptor structure.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERROR_NOT_OWNER`	Asserts in debug mode on illegal values.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts in debug mode on illegal values.

Description

Used to control the structure of a frame on the SSI bus, e.g., determining the characteristics of a frame on the SSI bus. It sets the MMIO registers with the ssiSetValidSlotSizeVSS and ssiSetFrameSizeFSS macros.

Implementation Notes

The **FSS** and **VSS** fields are changed.

ssiGetFraming

```
tmLibdevErr_t ssiGetFraming(
   Int               instance,
   pssiFrameSetup_t   frame
);
```

Parameters

`instance`	Instance variable assigned at **ssiOpen**.
`frame`	Pointer to frame descriptor structure.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERROR_NOT_OWNER`	Asserts in debug mode on illegal values.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts in debug mode on illegal values.

Description

Used to retrieve the structure of a frame on the SSI bus, e.g. determining the characteristics of a frame on the SSI bus. This is achieved by reading the MMIO registers with the ssiGetFrameSizeFSS and ssiGetValidSlotSizeVSS macros.

ssiStop

```
tmLibdevErr_t ssiStop(
   Int    instance
);
```

Parameters

`instance`	The instance, as returned by **ssiOpen**.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERROR_NOT_OWNER`	Asserts in debug mode on illegal values.

Description

ssiStop stops the audio transmit by disabling a transmission on the SSI bus. This is implemented as a call to the ssiDisable macro.

Implementation Notes

Stopping the SSI bus may necessitate a reset of the off chip peripheral. Interrupt enable bits are not affected by ssiStop.

ssiStart

```
tmLibdevErr_t ssiStart(
   Int    instance
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERROR_NOT_OWNER`	Asserts, in debug mode, on illegal values.

Description

Enables the interrupt sources requested in setup and starts the SSI transmission. After making the setup of IRQ with the macros **ssiEnableRIE**, **ssiEnableChangeDetectCDE**, **ssiEnableTIE**, this function enables the SSI with the **ssiEnable** macro.

ssiConfigure

```
tmLibdevErr_t ssiConfigure(
   Int                  instance,
   pssiInstanceSetup_t  setup
);
```

Parameters

`instance`	Instance, as returned by **ssiOpen**.
`setup`	Pointer to setup structure.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts in debug mode on illegal values.
`TMLIBDEV_ERROR_NOT_OWNER`	Asserts in debug mode on illegal values.
`BOARD_ERR_NULL_FUNCTION`	No config function is specified in the BSL.

Description

Gateway to a board support function that you supply. The **interruptLevelSelect**, **config-Buffer** and **configBufferLength** are passed to the configure function specified in the BSP (look at the Board API).

ssiOffHook

```
tmLibdevErr_t ssiOffHook(
   Int    instance
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERROR_NOT_OWNER`	Asserts, in debug mode, on illegal values.

Description

Used for telecom line control. If the board support package provides a hook function, it is called to pick up the phone line. If not, the IO2 bit is set low to go off hook with the macros ssiSetWIO2 and ssiSetModeIO2.

ssiOnHook

```
tmLibdevErr_t ssiOnHook(
   Int    instance
);
```

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERROR_NOT_OWNER`	Asserts, in debug mode, on illegal values.

Description

Used for telecom line control. If the board support package provides a hook function, it is called If not, the IO2 bit is set high to go on hook with the **ssiSetWIO2** and **ssiSetModeIO2** macros.

Chapter 2

UART API

Topic	Page
UART API Overview	28
Uart API Callback Functions	29
Uart API Data Structures	34
UART API Functions	43

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

Note
This API is preliminary and subject to change.

UART API Overview

The UART device library provides a TSA compatible interface to access serial interfaces connected to the TriMedia chip.

Before an application can use a serial port, it must open an instance of the port and initialize it. The application must also implement the callback functions required by the device library. The following code is an example of such a callback function.

```
typedef struct {
   _AppSem_Semaphore writeSemaphore;
   _AppSem_Semaphore readSemaphore;
}localScope_t;

localScope_t localScope = {
   _AppSem_INITIALISED_SEM,
   _AppSem_INITIALISED_SEM};

void myWriteCompletion( Int instance, Int count, Pointer handle ){
   localScope_t *s = (*localScope_t) handle;
   AppSem_V(s->writeSemaphore);
}
void myReadCompletion(Int instance, Int count, Pointer handle){
   localScope_t *s = (*localScope_t) handle;
   AppSem_V(s->readSemaphore);
}
   .
   .
   .
err = tsaUartOpen(&instance, UART_COM1);
CHECK(err); /* CHECK is an application macro that handles errors. */

setup.baudRate            = UART_BAUD_38400;
setup.numDataBits         = 8;
setup.numStopBits         = 1;
setup.parity              = UART_PARITY_NONE;
setup.controlMode         = UART_CONTROL_OFF;
setup.handle              = (Pointer) &localScope;
setup.writeCompletionFunc = myWriteCompletion;
setup.readCompletionFunc  = myReadCompletion;
setup.controlHandler      = Null;
setup.errorHandler        = Null;
err = tsaUartSetup(instance, &setup);
CHECK(err);
```

After the setup succeeded the application can write to the UART, thus:

```
/* initiate write */
   err = tsaUartWrite( instance, dataBuffer, dataSize );
   CHECK(err);
/* wait for completion function to release semaphore */
   AppSem_P( localScope.writeSemaphore );
```

It can also read from the UART, thus:

```
/* initiate read */
   err = tsaUartRead( instance, dataBuffer, dataSize );
   CHECK(err);
/* wait for completion function to release semaphore */
   AppSem_P( localScope.readSemaphore );
```

Uart API Callback Functions

This section presents the UART callback functions.

Name	Page
tsaUartErrorHandlerFunc_t	30
tsaUartWriteCompletionFunc_t	31
tsaUartReadCompletionFunc_t	31
tsaUartControlHandler_t	32
tsaUartConfigHandler_t	33

tsaUartErrorHandlerFunc_t

```
typedef void ( *tsaUartErrorHandlerFunc_t )(
   Int            instance,
   tmLibdevErr_t  err,
   Pointer        handle
);
```

Parameters

`instance`	Instance that called this function.
`err`	Error that occured.
`handle`	Handle passed to the UART library by the application.

Errors

`UART_ERR_TX_ERROR`	Error while transmitting data.
`UART_ERR_RX_OVERRUN_ERROR`	Overrun error while receiving data.
`UART_ERR_PARITY_ERROR`	Parity error.
`UART_ERR_FRAME_ERROR`	Framing error (no stopbit received).
`UART_ERR_BREAK_ERROR`	Received data were kept in silent state for a full word time, including the start bit, data bits, parity bit and stop bits.
`UART_ERR_OTHER_ERROR`	Another error occurred while sending or receiving data.
`UART_ERR_MULTIPLE_ERRORS`	Multiple errors have occurred.

Description

The UART library calls this error callback function from its interrupt handler to report errors that occur while transmitting or receiving data.

For errors returned by the API functions, see **tsaUartControlMode_t**.

tsaUartWriteCompletionFunc_t

```
typedef void ( *tsaUartWwriteCompletionFunc_t)(
   Int       instance,
   Int       count,
   Pointer   handle
);
```

tsaUartReadCompletionFunc_t

```
typedef void ( *tsaUartReadCompletionFunc_t )(
   Int       instance,
   Int       count,
   Pointer   handle
);
```

Parameters

`instance`	Instance that called the completion function.
`count`	Number of characters that have been written.
`handle`	Handle passed to the library by the application.

Description

The UART library uses non-blocking function calls to initiate reads and writes. To notify the application about the completion of those reads and writes, it uses completion call-back functions. The application must provide the completion functions. Pointers to the functions get passed to the library when you call **tsaUartInstanceSetup**.

The completion functions also get called when a read/write is aborted. See also **tsaUart-WriteAbort** and **tsaUartReadAbort**).

tsaUartControlHandler_t

```
typedef void (*tsaUartControlHandler_t)(
   Int               instance,
   tsaUartControl_t  event,
   Pointer           handle
);
```

Parameters

`instance`	Instance that called this function.
`event`	Control event that occurred.
`handle`	Handle passed to the UART library by the application.

Description

The UART library calls this function when a flow control event is detected. These are the possible events:

`FLOW_ON`	The connected device requests a pause in the flow of characters
`FLOW_OFF`	The connected device can accept more characters

This function is called in the same way for hardware and software flow control. (If hardware flow control is enabled, it is called when CTS changes value. If software flow control is enabled, it is called when an x-on or x-off character is received.)

tsaUartConfigHandler_t

```
typedef void (*tsaUartConfigHandler_t)(
   Int                    instance,
   tsaUartConfigEvent_t  event,
   Pointer               handle
);
```

Parameters

instance	Instance that called this function.
event	Config event that occurred.
handle	Handle passed to the UART library by the application.

Description

The UART library calls this function when a configuration event is detected. Thes are the possible events:

`UART_DSR_ON`	DSR goes high (“connected device is available”).
`UART_DSR_OFF`	DSR goes low (“connected device is not available”).
`UART_CTS_ON`	CTS goes high (“connected device can receive”).
`UART_CTS_OFF`	CTS goes low (“connected device is busy”).
`UART_RI_ON`	RI goes high (“incoming call ringing”).
`UART_RI_OFF`	RI goes low (“end of ring”).
`UART_DCD_ON`	DCD goes high (“call placed successfully”).
`UART_DCD_OFF`	DCD goes low (“call ended”).

If hardware flow control is enabled, the events **UART_CTS_ON** and **UART_CTS_OFF** do not cause this function to be called: instead, the controlHandler function is called with events **FLOW_ON** or **FLOW_OFF**. This enables an application to be written independently of the flow control regime. This type is a bitmask: more than one of the bits may be set when the configuration event callback is called.

Uart API Data Structures

This section presents the UART device library data structures.

Name	Page
tsaUartParity_t	35
tsaUartBaud_t	36
tsaUartConfig_t	37
tsaUartConfigEvent_t	38
tsaUartControl_t	39
tsaUartCapabilities_t	40
tsaUartInstanceSetup_t	41
tsaUartControlMode_t	42

tsaUartParity_t

```
typedef enum {
   UART_PARITY_NONE = 0,
   UART_PARITY_EVEN,
   UART_PARITY_ODD,
   UART_PARITY_MARK,
   UART_PARITY_SPACE
} tsaUartParity_t;
```

Fields

`UART_PARITY_NONE`	No parity bit is sent or expected.
`UART_PARITY_EVEN`	The parity bit is set or cleared so that the number of logic ones is even.
`UART_PARITY_ODD`	The parity bit is set or cleared so that the number of logic ones is odd.
`UART_PARITY_MARK`	The parity bit is always 1.
`UART_PARITY_SPACE`	The parity bit is always 0.

Description

This type sets the parity mode for the UART in the **tsaUartInstanceSetup_t** struct.

tsaUartBaud_t

```
typedef enum {
   UART_BAUD_9600   = 0x00000001,
   UART_BAUD_14400  = 0x00000002,
   UART_BAUD_19200  = 0x00000004,
   UART_BAUD_38400  = 0x00000008,
   UART_BAUD_57600  = 0x00000010,
   UART_BAUD_115200 = 0x00000020,
   UART_BAUD_230400 = 0x00000040,
   UART_BAUD_460800 = 0x00000080,
   UART_BAUD_921600 = 0x00000100
} tsaUartBaud_t;
```

Fields

`UART_BAUD_9600`	9600 baud.
`UART_BAUD_14400`	14,400 baud.
`UART_BAUD_19200`	19,200 baud.
`UART_BAUD_38400`	38,400 baud.
`UART_BAUD_57600`	57,600 baud.
`UART_BAUD_115200`	115,200 baud.
`UART_BAUD_230400`	230,400 baud.
`UART_BAUD_460800`	460,800 baud.
`UART_BAUD_921600`	921,600 baud.

Description

This type sets the baud rate for the UART in **tsaUartInstanceSetup_t**. It is also used in **tsaUartCapabilities_t** to report the hardware-supported baud rates.

tsaUartConfig_t

```
typedef enum {
   UART_RTS_ON
   UART_RTS_OFF
   UART_DTR_ON
   UART_DTR_OFF
   UART_GET_SCRATCH_REGI
   UART_SET_SCRATCH_REGI
   UART_SET_LOOP_BACK
} tsaUartConfig_t;
```

Fields

`UART_RTS_ON`	Sets RTS high ("this device can accept characters").
`UART_RTS_OFF`	Sets RTS low ("this device is busy").
`UART_DTR_ON`	Sets DTR high ("this device is available").
`UART_DTR_OFF`	Sets DTR low ("this device is not available").
`UART_GET_SCRATCH_REGI`	Read the scratch register.
`UART_SET_SCRATCH_REGI`	Write to the scratch register.
`UART_SET_LOOP_BACK`	Put the UART into loopback mode. (This mode can be used for debugging.)

Description

This type is used in **tsaUartInstanceConfig** to set the configuration command.

Although this function can be used to change the flow control line (RTS), normally you would use **tsaUartControl** for that.

tsaUartConfigEvent_t

```
typedef enum tsaUartConfigEvent_t {
   UART_DSR_ON  = 1 << 0,
   UART_DSR_OFF = 1 << 1,
   UART_CTS_ON  = 1 << 2,
   UART_CTS_OFF = 1 << 3,
   UART_RI_ON   = 1 << 4,
   UART_RI_OFF  = 1 << 5,
   UART_DCD_ON  = 1 << 6,
   UART_DCD_OFF = 1 << 7
} tsaUartConfigEvent_t;
```

Fields

`UART_DSR_ON`	DSR goes high (“connected device is available”).
`UART_DSR_OFF`	DSR goes low (“connected device is not available”).
`UART_CTS_ON`	CTS goes high (“connected device can receive”).
`UART_CTS_OFF`	CTS goes low (“connected device is busy”).
`UART_RI_ON`	RI goes high (“incoming call ringing”).
`UART_RI_OFF`	RI goes low (“end of ring”).
`UART_DCD_ON`	DCD goes high (“call placed successfully”).
`UART_DCD_OFF`	DCD goes low (“call ended”).

Description

This type is used in the config callback function.

tsaUartControl_t

```
typedef enum tsaUartControl_t {
   FLOW_OFF,
   FLOW_ON
} tsaUartControl_t;
```

Fields

`FLOW_ON`	Data flow is enabled.
`FLOW_OFF`	Data flow is paused.

Description

This type is used in **tsaUartControl** to pause or resume the flow of received characters; and also in the control callback function to indicate that the connected device has paused or resumed the flow of transmitted characters.

tsaUartCapabilities_t

```
typedef struc {
   tmVersion_t  version;
   Int          numSupportedInstances;
   UInt         numCurrentInstances;
   Char         name[DEVICE_NAME_LENGTH];
   UInt32       baudRates;
} tsaUartCapabilities_t, *ptsaUartCapabilities_t;
```

Fields

`version`	Version of the UART library.
`numSupportedInstances`	Number of supported instances.
`numCurrentInstances`	Number of instances currently in use.
`name`	Name of the UART (e.g., the UART chip in use).
`baudRates`	OR'd values of supported baud rates (of type **tsaUartBaud_t**).

Description

A structure of this type is used in **tsaUartGetCapabilities** to report the capabilities of a UART port.

tsaUartInstanceSetup_t

```
typedef struct{
   tsaUartBaud_t                  baudRate;
   Int                            numDataBits;
   Int                            numStopBits;
   tsaUartParity_t                parity;
   tsaUartControlMode_t           controlMode;
   Pointer                        handle;
   tsaUartWriteCompletionFunc_t   writeCompletionFunc;
   tsaUartReadCompletionFunc_t    readCompletionFunc;
   tsaUartControlHandler_t        controlHandler;
   tsaUartErrorHandlerFunc_t      errorHandlerFunc;
   tsaUartConfigHandler_t         configHandler;
}tsaUartInstanceSetup_t,*ptsaUartInstanceSetup_t;
```

Fields

`baudRate`	Baud rate.
`numDataBits`	Number of data bits: 5, 6, 7 or 8.
`numStopBits`	Number of stop bits: 1 or 2.
`parity`	Parity: odd, even, or none. See **tsaUartParity_t** on page 35 for acceptable values.
`controlMode`	Flow control mode.
`handle`	One parameter of the callback functions.
`writeCompletionFunc`	Write completion function, called by the library after a transmission has completed or after it has been aborted.
`readCompletionFunc`	Read completion function, called by the library after a reception has completed or after it has been aborted.
`controlHandler`	This function is called by the UART library if a flow-control event occurs. This field can be set to Null if the application does not want to do flow control.
`errorHandlerFunc`	This function is called if an error occurs. If this field is set to Null, an internal error function gets called instead.
`configHandler`	This function is called by the UART library when a configuration event occurs.

Description

A structure of this type is passed to **tsaUartInstanceSetup** to initialize a UART port.

tsaUartControlMode_t

```
typedef enum{
   UART_CONTROL_OFF,
   UART_CONTROL_SW,
   UART_CONTROL_HW
} tsaUartControlMode_t;
```

Fields

`UART_CONTROL_OFF`	No flow control.
`UART_CONTROL_SW`	Software flow control.
`UART_CONTROL_HW`	Hardware flow control.

Description

This type is used in the **tsaUartInstanceSetup_t** struct to set the mode for flow control. If the application wants to do flow control, then it also has to provide the **controlHandler** function.

UART API Functions

This section describes the UART device library's functional interface.

Name	Page
tsaUartGetNumberOfUnits	44
tsaUartGetCapabilities	45
tsaUartOpen	46
tsaUartInstanceSetup	47
tsaUartWrite	48
tsaUartRead	49
tsaUartWriteAbort	50
tsaUartReadAbort	51
tsaUartControl	52
tsaUartInstanceConfig	53
tsaUartClose	54

tsaUartGetNumberOfUnits

```
extern tmLibdevErr_t tsaUartGetNumberOfUnits(
   UInt32   *pNoOfCommPort
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts if **pNoOfCommPort** is a null pointer (but only in the debugging version).

Description

This function returns the number of UART ports available on the board on which the application is running.

tsaUartGetCapabilities

```
extern tmLibdevErr_t tsaUartGetCapabilities(
   unitSelect_t               portID,
   ptsaUartCapabilities_t    *caps
);
```

Parameters

`portID`	The port for which the application wants to get the capabilities.
`caps`	Pointer to a variable in which to return a pointer to the capabilities of the selected UART port.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW`	No UART is available in the hardware.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts if **caps** is a null pointer (but only in debugging version).

Description

This function gets the capabilities for a UART port.

tsaUartOpen

```
extern tmLibdevErr_t tsaUartOpen(
   Int           *instance,
   unitSelect_t   portID
);
```

Parameters

`instance`	Pointer to the (returned) instance. This instance must be used for subsequent calls to the API.
`portID`	UART port that will be opened.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBDEV_ERR_NOT_AVAILABLE_IN_HW`	The requested UART port is not available in the hardware.
`TMLIBDEV_ERR_NO_MORE_INSTANCES`	No more instances are available for the selected port.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts if **instance** is a null pointer (but only in the debugging version).
`TMLIBDEV_ERR_MEMALLOC_FAILED`	Memory allocation failed.

The function can also return error codes produced by the board support package. The function can assert **BOARD_ERR_NULL_FUNCTION** (in the debugging version) if the needed functions from the BSP are not available. The function can also return error codes produced by the PIC library.

Description

This function opens an instance of an UART port. All necessary resource are allocated. It also installs the interrupt handler using the PIC library.

tsaUartInstanceSetup

```
extern tmLibdevErr_t tsaUartInstanceSetup(
   Int                        instance,
   tsaUartInstanceSetup_t   *setup
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts if **setup** is a null pointer (but only in the debugging version).
`TMLIBDEV_ERR_NOT_OWNER`	Asserts if **instance** is invalid (but only in the debugging version).
`UART_ERR_NO_TX_COMPLETION_FUNC`	Asserts if no read completion function is supplied (but only in the debugging version).
`UART_ERR_NO_RX_COMPLETION_FUNC`	Asserts if no write completion function is supplied (but only in the debugging version).
`UART_ERR_INVALID_BAUDRATE`	Invalid baud rate.
`UART_ERR_INVALID_NUM_DATA_BITS`	Invalid number of data bits.
`UART_ERR_INVALID_PARITY`	Invalid parity setting.
`BOARD_ERR_NULL_FUNCTION`	No init function in BSP.
`TMLIBDEV_ERR_MEMALLOC_FAILED`	Memory allocation failed.

The function can also return error codes produced by the board support package and the PIC library.

Description

Initializes an instance of the UART port.

tsaUartWrite

```
extern tmLibdevErr_t tsaUartWrite(
   Int        instance,
   Address    buffer,
   UInt32     size
);
```

Parameters

`instance`	Instance previously opened by `tsaUartOpen`.
`buffer`	Pointer to a buffer containing data that will be sent to the UART.
`size`	Size of the buffer.

Return Codes

`TMLIBAPP_OK`	Success.
`UART_ERR_INVALID_SIZE`	Size is 0.
`UART_ERR_WRITE_DEVICE_IN_USE`	The previous write has not finished.
`TMLIBDEV_ERR_NOT_OWNER`	Asserts if the instance is invalid (but only in the debugging version).
`UART_ERR_NO_SETUP`	Asserts if UART port has not been initialized (but only in the debugging version).
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts if **buffer** is a null pointer (but only in the debugging version).

The function can also return error codes produced by the board support package.

Description

Writes a buffer to the UART. This function is not a blocking function; it returns immediately. When the write is finished, the UART library calls the application's write completion function.

The application can stop the transmission by calling **tsaUartWriteAbort**.

tsaUartRead

```
extern tmLibdevErr_t tsaUartRead (
   Int       instance,
   Address   buffer,
   UInt32    size
);
```

Parameters

`instance`	Instance, as returned by **tsaUartOpen**.
`buffer`	Pointer to a buffer that will receive the data.
`size`	Size of the buffer.

Return Codes

`TMLIBAPP_OK`	Success.
`UART_ERR_INVALID_SIZE`	Size is 0.
`UART_ERR_READ_DEVICE_IN_USE`	The previous read has not been finished.
`TMLIBDEV_ERR_NOT_OWNER`	Asserts if the instance is invalid (but only in the debugging version).
`UART_ERR_NO_SETUP`	Asserts if UART port has not been initialized (but only in the debugging version).
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts if **buffer** is a null pointer (but only in the debugging version).

The function can also return error codes produced by the board support package.

Description

This function transfers data from the UART to a buffer.

This function is not a blocking function; it returns immediately. After receiving the specified number of characters, the UART library calls the application's read completion function.

The application can stop the receive process by calling **tsaUartReadAbort**.

tsaUartWriteAbort

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

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Asserts if the instance is invalid (but only in the debugging version).

The function can also return error codes produced by the board support package.

Description

This function aborts a running write operation.

tsaUartReadAbort

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

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Asserts if the instance is invalid (but only in the debugging version).

The function can also return error codes produced by the board support package.

Description

This function aborts a running read operation.

tsaUartControl

```
extern tmLibdevErr_t tsaUartControl(
   Int                instance,
   tsaUartControl_t   command
);
```

Parameters

`instance`	Instance previously opened by **tsaUartOpen**.
`command`	Control command.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function sends a control command to the UART port.

tsaUartInstanceConfig

```
extern tmLibdevErr_t tsaUartInstanceConfig(
   Int               instance,
   tsaUartConfig_t   command,
   Pointer           value
);
```

Parameters

`instance`	Instance, as returned by **tsaUartOpen**.
`command`	Configuration command.
`value`	Pointer to the configuration value.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Asserted if **instance** is invalid (but only in the debugging version).

The function can also return error codes produced by the board support package.

Description

This function configures the UART.

Because it passes the command and value directly to the board support package, it can also be used to implement features of UARTs that are not supported by the current implementation.

tsaUartClose

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

Parameters

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

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Asserts if **instance** is invalid (but only in the debugging version).

The function can also return error codes produced by board support package or PIC device library.

Description

This function closes an instance of a UART port. All running writes and reads are aborted and all allocated resources are freed.

To use this UART port again, you must reopen it using **tsaUartOpen**.

Chapter 3

V34 Modem API

Topic	Page
Overview	56
TriMedia V34 Modem API Overview	57
Interfacing V42 with the Modem	59
Modem Data Structures	60
Modem Functions	72
I/O Handlers	87
Status Handlers	90
Modem Example: exModem.c	92
PPP Example: exModemPPP.c	94
Known Bugs	95

Disclaimer
The API described in this document is pre-alpha. Changes in the API, in the data structure, and in the I/O and status handlers are expected in the next release.

Overview

The V34/V34bis is a standard for reliable data transmission using plain old telephone service (POTS). The V34bis standard is usually implemented with a V25Ter front-end (AT command set) and with the V42/V42bis standard for compression and error correction.

The TM softmodem V34bis API data library comprises the following modules:

- Dialer. Issues commands to the analog front end (AFE). Puts the modem onhook or offhook, generates a dialing signal, and detects tones such as busy signal. It is controlled by the V25Ter module.
- V34bis Datapump. Implements all the modules for V34/V34bis functionality, including modulation, demodulation, equalization and echo cancellation. The V34bis datapump interfaces with the AFE through internal interrupt service routines (ISRs). The V34bis datapump described in Figure 1 has V32/V22 functionality to keep backward compatility with previous ITU modem standards.
- V25Ter. AT command set. Issues commands to the AFE, V34 datapump and V42 module to handle dialing, call answering, and mode settings.
- V42. Provides V42/V42bis for error correction and compression.

Figure 1 illustrates these modules. The solid arrows represent data flow and the dashed arrows represent control signals.

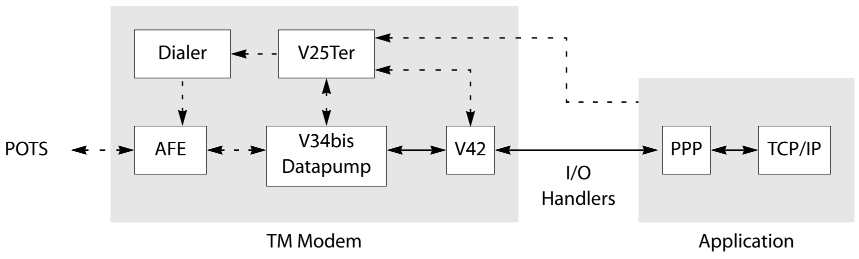


Figure 1 Structure of TM Modem

An application, such as a TCP/IP stack over PPP, can be implemented with the use of the TM V34 softmodem API, as shown in Figure 1. The application interfaces with the modem, using AT commands, by calling an API function that sends commands to the V25Ter module. The V25Ter module interacts with the V34bis Datapump, dialer and V42 module to change mode settings and to handle dialing and answering. I/O handlers supplied by the application perform the data transfer between the application and the V34 modem.

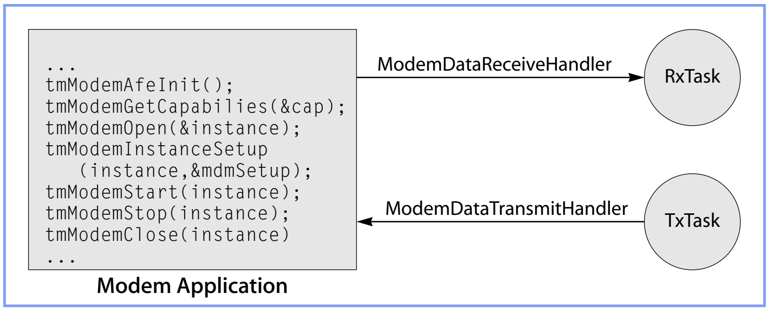
TriMedia V34 Modem API Overview

This version of the TriMedia V34 Modem API supports the Philips TriMedia TM-1100 Reference Platform and the Philips DTV Reference Platform. This release of the V34 Modem API supports only the AFEs based on the STL7545 originally present on these plaforms.

A typical modem application would use the following sequence of API calls to initialize and run the modem:

- **tmModemAfeInit( )** Initialize supported platforms.
- **tmModemGetCapabilities(&cap)** Obtain modem capabilities structure.
- **tmModemOpen(&instance)** Open a modem instance. The current version supports one instance.
- **tmModemInstanceSetup(instance, &mdmSetup)** Initialize the modem instance according to the settings present in the **mdmSetup** structure.
- **tmModemStart(instance)** Start the modem.
- **tmModemStop(instance)** Stop datapump processing.
- **tmModemClose(instance)** Release an instance of the modem and free memory allocated for internal variables.

The receive and transmit tasks use I/O handlers defined by the application. The **ModemDataReceiveHandler** is invoked whenever the modem has finished processing data received from the AFE. The **ModemDataTransmitHandler** is invoked when modem is ready to accept data to be processed and sent to the AFE. The I/O handlers are identified in the root application by the **tmModemInstanceSetup_t** structure. Figure 2 illustrates the scheme, where "RxTask" and "TxTask" represent the receive and transmit tasks.



```
...
tmModemAfeInit();
tmModemGetCapabilies(&cap);
tmModemOpen(&instance);
tmModemInstanceSetup
   (instance,&mdmSetup);
tmModemStart(instance);
tmModemStop(instance);
tmModemClose(instance)
...
```

Figure 2 Modem application interfaced with the receive and transmit tasks by I/O handlers.

Note that the I/O handlers are not invoked by the modem application directly or by the Rx/Tx tasks. The I/O handlers are to be called by the modem only when it needs to obtain input data or to release output data.

When the V42 module is active, the I/O handlers and the receive and transmit tasks can use buffer-handling routines supplied by the V34 Modem API for exchanging data. These functions are:

- **tmModemV42PutCharInRxBuffer**. This function should be used by the receive handler to place a character in the receive buffer.
- **tmModemV42GetCharFromRxBuffer**. This function should be used by the receive task to get a character from the receive buffer.
- **tmModemV42PutCharInTxBuffer**. This function should be used by the transmit task to place a character in the transmit buffer. .
- **tmModemV42GetCharFromTxBuffer**. This function should be used by the transmit handler to receive a character fom the transmit buffer.

Figure 3 illustrates these functions:

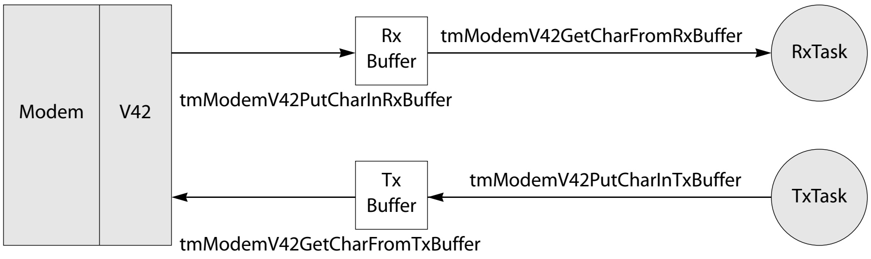

Modem
V42
Rx Buffer
tmModemV42PutCharInRxBuffer
tmModemV42GetCharFromRxBuffer
RxTask
Tx Buffer
tmModemV42PutCharInTxBuffer
TxTask
tmModemV42GetCharFromTxBuffer

Figure 3 Buffer handling routines used in the I/O handlers and transmit/receive tasks

The application can send commands to the modem to place and answer calls using V25Ter/AT command syntax. For example, the application can obtain an AT command string from a terminal, and send it to the modem by calling **tmModemV25Send-Command**.

The V42 module interfaces with the modem using specific API calls used in the modem status handlers, described in the following section.

Interfacing V42 with the Modem

The receive and transmit applications and the I/O handlers must handle cases when the V42 module is on or off. To turn V42 on or off, set the **V42ModuleActive** field **TRUE** or **FALSE**, respectively, in the **tmModemInstanceSetup_t** structure passed to **modemInstance-Setup**.

The status handler must execute some commands during particular states of the modem and V42 status handlers. These commands are necessary to restart the V42 module once a modem connection is established, to pause the V42 processing during retrains and restarts, to generate empty frames, and to stop the modem if the V42 module detects dis-connection. In particular:

- Handler: **ModemLocalStatusHandler**
  Status: **tmModemConnected**
  If the V.42 module is active, the following commands must be issued:

```
tmModemV42Command_t cmd;
cmd.params.dataRate = modemStatus.state.connect.dataRate;
cmd.command = tmModemV42RestartCmd;
tmModemV42CommandHandler(&cmd);
```

- Handler: **ModemLocalStatusHandler**
  Status: **tmModemRetrainStarted** and **mRateRenegotiated**
  The following commands must be issued:

```
tmModemV42Command_t cmd;
cmd.command = tmModemV42PauseCmd;
tmModemV42CommandHandler(&cmd);
```

- Handler: **V42LocalStatusHandler**
  Status: **tmModemV42HandShakeInProgress, vConnectedCompression** and **tmModemV42ConnectedErrorControl**.
  The following commands must be issued:

```
tmModemCommand_t cmd;
cmd.param.emptyFrame = tmModemHDLCFrame;
cmd.command          = tmModemSetEmptyFrameCmd;
tmModemCommandHandler(&cmd);
```

- Handler: **V42LocalStatusHandler**
  Status: **tmModemV42DisconnectedState**.
  The following commands must be issued.

```
tmModemCommand_t cmd;
cmd.command = tmModemStopCmd;
tmModemCommandHandler(&cmd);
```

Modem Data Structures

This section presents the V34 modem library data structures.

Name	Page
tmModemCallMode_t	61
tmModemCommandCode_t	62
tmModemCapabilities_t	63
tmModemStatus_t	65
tmModemStatusCode_t	66
tmModemV42Command_t	67
tmModemV42CommandCode_t	68
tmModemInstanceSetup_t	69
tmModemV8BisConfig_t	70
tmModemV8BisStatus_t	70
tmModemV8Protocol_t	71

tmModemCallMode_t

```
typedef enum {
   tmModemAnswerMode,
   tmModemCallerMode,
   tmModemInitiatorMode
} tmModemCallMode_t;
```

Fields

`tmModemAnswerMode`	Used by V25Ter to put the modem in answering mode.
`tmModemCallerMode`	Used by V25Ter to put the modem in calling mode.
`tmModemInitiatorMode`	*Reserved*.

Description

Enumerates values used by the dialer module to put the mode in calling or answering mode.

tmModemCommandCode_t

```
typedef enum {
   tmModemStartCmd,
   tmModemStopCmd,
   tmModemStartClearDownCmd,
   tmModemSetStatusHandlerCmd,
   tmModemSetEmptyFrameCmd,
   tmModemSetDesiredModeCmd,
   tmModemSetDteCapabilitiesCmd,
   tmModemSetDataReadHandlerCmd,
   tmModemSetDataWriteHandlerCmd,
   tmModemSetV8_bisParametersCmd,
   tmModemSetTotalHardwareDelayCmd,
   tmModemGetConnectionDetailsCmd,
   tmModemForceRetrainCmd,
   tmModemForceRenegotiationCmd,
   tmModemSetConstellationHandlerCmd
} tmModemCommandCode_t;
```

Fields

`tmModemStartCmd`	Used to start the modem.
`tmModemStopCmd`	Used to stop the modem, if an escape code is issued or if carrier is dropped.
`tmModemStartClearDownCmd`	Used to start clear down procedure.
`tmModemSetStatusHandlerCmd`	Used to set modem status handler.
`tmModemSetEmptyFrameCmd`	Used to set emply frame.
`tmModemSetDesiredModeCmd`	*Reserved.*
`tmModemSetDteCapabilitiesCmd`	*Reserved.*
`tmModemSetDataReadHandlerCmd`	Used to set modem data read handler.
`tmModemSetDataWriteHandlerCmd`	Used to set modem data write handler.
`tmModemSetV8_bisParametersCmd`	*Reserved.*
`tmModemSetTotalHardwareDelayCmd`	Used to set the total hardware delay.
`tmModemGetConnectionDetailsCmd`	*Reserved.*
`tmModemForceRetrainCmd`	*Reserved.*
`tmModemForceRenegotiationCmd`	*Reserved.*
`tmModemSetConstellationHandlerCmd`	*Reserved.*

Description

These values are used to issue commands to the modem command handler.

tmModemCapabilities_t

```
typedef struct {
   tsaDefaultCapabilities_t  *default_cap;
   capsStruct                *Caps;
   Int                        v42ModuleActive;
   Int                        numCurrentInstances;
} tmModemCapabilities_t;
```

Fields

`default_cap`	Pointer to default capabilities structure.
`Caps`	Pointer to capabilities structure.
`v42ModuleActive`	Integer with non-zero value if V42 is active.
`numCurrentInstances`	Number of instances being used.

Description

Holds a list of capabilities. The V34 Modem maintains a structure of this type to describe itself. The application can retrieve the address of this structure by calling **modemGet-Capabilities**.

tmModemV25Capabilities_t

```
typedef struct {
   Int32  dteMask;
   Int32  dceMask;
   Int32  symMask;
   Int32  rateMask;
   Int32  miscMask;
   Float  levelTX;
} tmModemV25Capabilities_t;
```

Fields

`dteMask`	DTE mask.
`dceMask`	DCE mask.
`symMask`	Reserved.
`rateMask`	Data rate mask.
`miscMask`	Reserved.
`levelTX`	Transmit level for hardware adjustment.

Description

Structure with masks used by the **tmModemCapabilities_t** structure.

tmModemStatus_t

```
typedef struct {
   tmModemStatusCode_t  status;
   struct {
      struct {
         Int32 modulation;
         Int32 dataRate;
         Int32 symbolRate;
      } connect;
      struct {
         Int32 remoteDteCapabilities;
         Int32 remoteDceCapabilities;
         Int32 remoteDteDesiredMode;
         Int32 remoteDceDesiredMode;
         Int time;
         tmModemV8CallFunction_t     callfØ;
         tmModemV8Protocol_t protØ;
         tmModemV8BisStatus_t status;
      } V8_bis;
      Int sampleFrequency;
      tmModetmModemConnectionDetails_t  connDetails;
      tmModetmModemExitCode_t exitCode;
   } state;
} tmModemStatus_t;
```

Fields

`status`	Modem status. See **tmModemStatusCode_t**.
`modulation`	Codes with allowed modem modulation types. Possible values, defined in tmModem.h, are:

```
MODEM_DTE_PROTOCOL_DATA     ØxØØØØØØØ1L
MODEM_DCE_PROTOCOL_ALL      ØxFFFFFFFFL
MODEM_DCE_PROTOCOL_V21      ØxØØØØØØØ1L
MODEM_DCE_PROTOCOL_V22      ØxØØØØØØØ2L
MODEM_DCE_PROTOCOL_V22BIS   ØxØØØØØØØ4L
MODEM_DCE_PROTOCOL_V23      ØxØØØØØØØ8L
MODEM_DCE_PROTOCOL_V32      ØxØØØØØØ1ØL
MODEM_DCE_PROTOCOL_V32BIS   ØxØØØØØØ2ØL
MODEM_DCE_PROTOCOL_V34BIS   ØxØØØØØØ4ØL
```

`dataRate`	Modem data rate.
`symbolRate`	Modem symbol rate.
`V8_bis`	Reserved for future use.
`sampleFrequency`	Sampling frequency used in datapump.
`connDetails`	Reserved for internal use.
`exitCode`	Returned modem exit code. See **tmModetmModemExitCode_t**.

Description

Structure used to verify modem status in the modem status handler.

tmModemStatusCode_t

```
typedef enum {
   tmModemExit,
   tmModemBusy,
   tmModemCarrierPresent,
   tmModemCarrierLost,
   tmModemConnected,
   tmModemSampleFrequency,
   tmModemV8Info,
   tmModemV8bisInfo,
   tmModemRateRenegotiationStarted,
   tmModemRetrainStarted,
   tmModemClearDownStarted,
   tmModemConnectionDetails
} tmModemStatusCode_t;
```

Fields

`tmModemExit`	Indicates that the modem has exited.
`tmModemBusy`	Indicates busy signal.
`tmModemCarrierPresent`	Indicates that a carrier is beging received.
`tmModemCarrierLost`	Indicates that carrier was lost.
`tmModemSampleFrequency`	Indicates the used sampling frequency.
`tmModemV8Info`	*Reserved for future use*.
`tmModemV8bisInfo`	*Reserved for future use*.
`tmModemRateRenegotiationStarted`	Indicates start of rate renegotiation.
`tmModemRetrainStarted`	Indicates start of retrain.
`tmModemClearDownStarted`	Indicates start of clear down procedure.
`tmModemConnectionDetails`	*Reserved*.

Description

These values contain status codes generated by the modem datapump.

tmModemV42Command_t

```
typedef struct tmModemV42Command_t {
   tmModemV42CommandCode_t               command;
   union {
      tmModemV42OptionalParam_t          opParams;
      tmModemV42StartParam_t             startParams;
      tmModemV42LockState_t              lockState;
      tmModemV42BreakSignalParam_t       breakSignalParams;
      tmModemV42DatapumpFunctions_t      dpFunctions;
      Int                                dataRate;
      struct {
         tmtmModemV42DataReadHandler_t   dataReadHandlerPtr;
         Int                             blockSize;
      }dataReadIO;
      tmModemV42DataWriteHandler_t       dataWriteHandlerPtr;
      tmModemV42StatusHandler_t          statusHandlerPtr;
   }params;
}tmModemV42Command_t;
```

Fields

`command`	Command to be sent to V42 module.
`opParams`	*Reserved.*
`startParams`	*Reserved.*
`lockState`	*Reserved.*
`breakSignalParams`	*Reserved.*
`dpFunctions`	*Reserved.*
`dataRate`	Data rate. Should be initialized to the current datarate when the command **tmModemV42RestartCmd** is sent to the V42 module. The current datarate can be obtained from **modemStatus.state.connect.datarate**.
`dataReadIO`	*Reserved.*
`dataWriteHandlerPtr`	*Reserved.*
`statusHandlerPtr`	*Reserved.*

Description

The first field of this structure contains the command to be sent to the V42 module. The other parameters contain fields that are used internally upon issuing commands to the V42 module.

tmModemV42CommandCode_t

```
typedef enum tmModemV42CommandCode_t {
   tmModemV42StartCmd,
   tmModemV42StopCmd,
   tmModemV42PauseCmd,
   tmModemV42RestartCmd,
   tmModemV42SetOpParametersCmd,
   tmModemV42SetDataReadHandlerCmd,
   tmModemV42SetDataWriteHandlerCmd,
   tmModemV42SetStatusHandlerCmd,
   tmModemV42SendBreakCmd,
   tmModemV42SetBreakParametersCmd,
   tmModemV42GetConnectionDetailsCmd,
   tmModemV42DataDeliveryCmd,
   tmModemV42ResetTimeReferenceCmd,
   tmModemV42SetDatapumpFunctionsCmd
}tmModemV42CommandCode_t;
```

Fields

`tmModemV42StartCmd`	Command to start V42 module
`tmModemV42StopCmd`	Command to stop V42 module.
`tmModemV42PauseCmd`	Command to pause v42 during modem retrain
`tmModemV42RestartCmd`	Command to perform restarting after pausing.
`tmModemV42SetOpParametersCmd`	*Reserved.*
`tmModemV42SetDataReadHandlerCmd`	*Reserved.*
`tmModemV42SetDataWriteHandlerCmd`	*Reserved.*
`tmModemV42SetStatusHandlerCmd`	*Reserved.*
`tmModemV42SendBreakCmd`	Command to send a BREAK to v42.
`tmModemV42SetBreakParametersCmd`	Command to set BREAK Parameters for v42.
`tmModemV42GetConnectionDetailsCmd`	Command to get V42 connection information.
`tmModemV42DataDeliveryCmd`	Command to start/stop v42 data dilevery.
`tmModemV42ResetTimeReferenceCmd`	Command to Reset the time.
`tmModemV42SetDatapumpFunctionsCmd`	Command to set the data pump functions.

Description

These values contain command codes used by **tmModemV42CommandHandler.**

tmModemInstanceSetup_t

```
typedef struct {
    modemDataWriteHandlerType   ModemDataTxHandler;
    modemDataReadHandlerType    ModemDataRxHandler;
    modemStatusHandlerType      ModemStatusHandler;
    tmModemV42StatusHandler_t   V42StatusHandler;
    Bool                        V42ModuleActive;
} tmModemInstanceSetup_t;
```

Fields

`ModemDataTxHandler`	Pointer to transmit handler.
`ModemDataRxHandler`	Pointer to receive handler.
`ModemStatusHandler`	Pointer to handler that manages status information about the modem. Typically, it uses information from **tmModemStatusCode_t**. See section about modem status handlers.
`V42StatusHandler`	Pointer to handler that manages status information about the V42 module. Typically, it uses information from **tmModemV42Status_t**. See section about modem status handlers.
`V42ModuleActive`	Variable of type **Bool**. It should be set to TRUE if V42 is chosen to be active and FALSE otherwise.

Description

This structure contains the modem handlers used for transmitting and receiving data, and also the status handlers for the modem and the V42 module. In addition, it contains one boolean variable that determines if V42 is active or not. It is used by the function **tmModemInstanceSetup** to initialize a modem instance properly.

tmModemV8BisConfig_t

```
typedef struct {
   UInt getRemoteCapabilities  : 1;
   UInt remoteHasV8bis         : 1;
   UInt mscLocal               : 1;
   UInt modeAcceptanceCriteria : 1;
   UInt retransmitCounterLimit : 2;
   UInt unused                 :10;
} tmModemV8BisConfig_t;
```

Description

Reserved for future use.

tmModemV8BisStatus_t

```
typedef struct {
   UInt   sessionResult     : 1;
   UInt   timerExpired      : 1;
   UInt   modeSelectionFail : 1;
   UInt   ansamDetected     : 1;
   UInt   ansDetected       : 1;
   UInt   transactionError  : 1;
   UInt   callerMode        : 1;
   UInt   unused            : 9;
} tmModemV8BisStatus_t;
```

Description

Reserved for future use.

tmModemV8Protocol_t

```
typedef enum {
   tmModemV8OctetProtocolLAPM = 0x2A,
   tmModemV8OctetProtocolExt  = 0xEA
} tmModemV8Protocol_t;
```

Description

Reserved for future use.

Modem Functions

This section presents the V34 Modem library functions.

Name	Page
tmModemAfeInit	73
tmModemOpen	73
tmModemClose	74
tmModemV25SendCommand	75
tmModemCommandHandler	76
tmModemGetCapabilities	76
tmModemInstanceSetup	77
tmModemDataWrite	78
tmModemStart	79
tmModemStop	79
tmModemV42Process	80
tmModemV42DataRead	80
tmModemV42DataWrite	81
tmModemV42DataReadPending	81
tmModemV42DataWritePending	82
tmModemV42CommandHandler	82
tmModemV42GetCharFromTxBuffer	83
tmModemV42PutCharInTxBuffer	84
tmModemV42GetCharFromRxBuffer	85
tmModemV42PutCharInRxBuffer	86

tmModemAfeInit

```
tmLibappErr_t tmModemAfeInit(void);
```

Description

Initializes the analog front end. Must be called before **tmModemOpen**.

tmModemOpen

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

Parameters

`instance`	Pointer (returned) to an instance variable, used to identify the instance in subsequent transactions.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Creates an instance of a V34 modem and sets the instance variable. This instance variable must be used in subsequent function calls for this modem. The open function allocates memory for the internal instance variables. The current version of this API only allows one instance.

tmModemClose

```
tmLibappErr_t tmModemClose(
   Int   instance
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number.

Description

This function releases the instance of the modem. It frees the memory allocated for internal variables.

tmModemV25SendCommand

```
tmLibappErr_t tmModemV25SendCommand(
   UInt32  instance,
   String  cmd
)
```

Parameters

`cmd`	Pointer to chars containing V25Ter commands supported by the TM V34 softmodem API.
`instance`	Modem instance being used to send V25Ter command.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Executes V25Ter (AT) command in **cmd**. Supported AT commands in this release are shown in Table 1. Additional commands are planned for the next release of the TM modem.

Table 1 Supported AT command set (V25Ter).

ATDT	Dials a phone number.
ATS0=X	Connects upon receiving a call after ringing X times.
AT&F1	Restores factory settings.
ATI	Obtains information about the modem.
ATZ	Restarts the modem.
ATH	Hangs up the phone connection.
A/	Repeats previous command.
AT	Same as ATZ.

tmModemCommandHandler

```
tmLibappErr_t tmModemCommandHandler(
   tmModemCommand_t  *cmd
);
```

Parameters

`cmd`	Pointer to the command.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Processes commands sent to datapump. Returns value with error code. See **tmModemCommand_t**.

tmModemGetCapabilities

```
tmLibappErr_t tmModemGetCapabilities(
   ptmModemCapabilities_t  *cap
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.

Description

This function can be used to retrieve a pointer to the capabilities struct of the TriMedia V34 Modem library. For more information, refer to **tmModemCapabilities_t** on page 63.

tmModemInstanceSetup

```
tmLibappErr_t tmModemInstanceSetup(
   Int                       instance,
   tmModemInstanceSetup_t  *setup
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number.

Description

Initializes V42 module if V42 is chosen to be active and installs modem I/O handlers (**ModemDataReceiveHandler** and **ModemDataTransmitHandler**). It resets the modem and the V42 module.

tmModemDataWrite

```
Int tmModemDataWrite(
   Int    nBytes,
   Byte  *srcPtr
);
```

Parameters

nBytes	Number of bytes to be sent to datapump.
scrPtr	Pointer to data to be sent to datapump.

Return

(Int)	An integer containing the number of bytes actually stored in the buffer. It may be less than **nBytes** if buffer overflow happens.

Description

This function stores **nBytes** of given data into the internal modem buffer for later transmission. The return value is the number of bytes actually stored in the buffer and may be less than **nBytes** if not enough space is available. Buffer overflow can be avoided by calling **tmModemDataWritePending** before **tmModemDataWrite**. This API call should only be used inside I/O handlers.

tmModemStart

```
tmLibappErr_t tmModemStart (
   Int instance
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	`Success.`
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number.

Description

Initializes V42 module if requested to be active, sends commands to V25Ter module and creates a modem task.

tmModemStop

```
tmLibappErr_t tmModemStop (
   Int   instance
);
```

Parameters

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

Return Codes

`TMLIBAPP_OK`	`Success.`
`TMLIBAPP_ERR_INVALID_INSTANCE`	Invalid instance number.

Description

The function stops datapump processing.

tmModemV42Process

```
tmLibappErr_t tmModemV42Process ( void );
```

Return Codes

V42 Exit Code.

Description

Executes the main V42 process.

tmModemV42DataRead

```
tmLibappErr_t tmModemV42DataRead(
   Int    nBytes,
   Byte  *RcvPtr
);
```

Parameters

`nBytes`	Number of bytes to be read from V42 buffer.
`RcvPtr`	Pointer to data to be copied from V42 buffer.

Description

Copies **nBytes** of data from V42 read buffer.

tmModemV42DataWrite

```
tmLibappErr_t tmModemV42DataWrite(
   Int    nBytes,
   Byte  *xmtPtr
);
```

Parameters

`nBytes`	Number of byes to be transmitted to the V42 module.
`xmtPtr`	Pointer to bytes to be transmitted to the V42 module.

Return Codes

Variable of type **tmModemV42ReturnErrorCode_t** containing return errors.

Description

Transfers **nBytes** bytes from data location pointed by **xmtPtr** to V42 module, that will be later sent to the datapump. This API call should only be used inside I/O handlers; a direct call inside the root task will cause incorrect operation.

tmModemV42DataReadPending

```
Int tmModemV42DataReadPending ( void );
```

Return

`(Int)`	Number of bytes in the V.42 read buffer in number of bytes.

Description

Before the Modem can deliver data to V.42, it must poll the status of the V.42 receive buffer for empty space by a call to **tmModemV42DataReadPending**. The function returns the empty space in the buffer (in bytes).

tmModemV42DataWritePending

```
Int tmModemV42DataWritePending ( void );
```

Return

`(Int)`	Returns the number of bytes in V42 internal buffer.

Description

Before the application can send data to V42, it must poll the status of the V42 receive buffer. This is accomplished by a call to the function **tmModemV42DataWritePending**. The function returns the empty space in the buffer in number of bytes.

tmModemV42CommandHandler

```
tmLibappErr_t tmModemV42CommandHandler(
   tmModemV42Command_t  *cmd
);
```

Parameters

`cmd`	Pointer to structure containing V42 command to be executed.

Return Codes

Variable containing error code.

tmModemV42GetCharFromTxBuffer

```
UInt32 tmModemV42GetCharFromTxBuffer(
   UInt32   length,
   Byte    *Buffer
);
```

Parameters

`length`	Number of characters to be obtained from transmit buffer.
`Buffer`	Pointer to character with locations to be used to obtain data from transmit buffer.

Return

`(UInt32)`	Number of characters successfully copied.

Description

Copies **length** characters from transmit buffer into memory locations pointed by **Buffer**. Should be used by the modem receive handler to obtain data from the buffer between the V42 module and application.

tmModemV42PutCharInTxBuffer

```
Int tmModemV42PutCharInTxBuffer(
   Int    length,
   Byte  *Buffer
);
```

Parameters

`length`	Number of characters to be copied into transmit buffer.
`Buffer`	Pointer to character with locations to be used to copy data into transmit buffer.

Return

`(Int)`	Number of characters successfully copied into transmit buffer.

Description

Copies **length** characters from memory locations pointed by **Buffer** into transmit buffer. Should be used by the application to copy data into the transmit buffer between the application and the V42 module.

tmModemV42GetCharFromRxBuffer

```
UInt32 tmModemV42GetCharFromRxBuffer(
   UInt32   length,
   Byte     *databyte
);
```

Parameters

`length`	Integer containing number of characters to be copied to memory locations pointed by databyte from the receive buffer.
`databyte`	Pointer to the receiver buffer.

Return

`(UInt32)`	Number of bytes successfully copied.

Description

Copies **length** elements from the modem receive buffer into memory locations pointed by **databyte**. Should be used by the application to receive data from the receive buffer between the V42 module and the application.

tmModemV42PutCharInRxBuffer

```
Int tmModemV42PutCharInRxBuffer(
   Int     length,
   Byte   *Buffer
);
```

Parameters

`length`	Integer containing number of characters to be copied from memory locations pointed by Buffer into the receive buffer.
`Buffer`	Pointer to the receive buffer.

Return

`(Int)`	The number of elements, of type **char**, successfully copied into the receive buffer.

Description

Copies **length** elements pointed by **Buffer** into the modem receive buffer. Should be used by the modem receive handler to copy data into the receive buffer between the V42 module and the application.

I/O Handlers

Modem input and output is performed by I/O handlers provided by the application. Two I/O handlers are needed: **ModemDataReceiveHandler** and **ModemDataTransmitHandler**. The I/O handlers are installed by calling **ModemInstanceSetup**, which requires an input parameter of type **tmModemInstanceSetup_t** containing pointers to the I/O handlers.

Name	Page
ModemDataReceiveHandler	88
ModemDataTransmitHandler	89

ModemDataReceiveHandler

```
UInt32 ModemDataReceiveHandler(
   Int     nBytes,
   Byte   *data
);
```

Parameters

`nBytes`	Number of bytes available at receive buffer.
`data`	Pointer to memory locations from receive buffer.

Description

ModemDataReceiveHandler must be supplied by the application to handle the interface between the modem output and the application's input. If the V42 module is not active, send data to the application by copying **nBytes** starting at address **data**. If V42 module is active, use **tmModemV42PutCharInRxBuffer** to put data in the buffer between V42 and the application (see *Interfacing V42 with the Modem* on page 59). Figure 4 shows an example.

```
UInt32
ModemDataReceiveHandler(
   int   nBytes,
   Byte *data
){
   int i, dataWriteInAppRxBuff;

   if( !v42ModuleActive ){
      for( i=0; i<nBytes; i++ ) dest[i] = data[i];
      destnBytes = nBytes;
   }else{
      dataWriteInAppRxBuff = tmModemV42PutCharInRxBuffer( nBytes, data );
      return( dataWriteInAppRxBuff );
   }
}
```

Figure 4 Code example for ModemDataReceiveHandler.

ModemDataTransmitHandler

```
void ModemDataTransmitHandler( void );
```

Description

ModemDataTransmitHandler must be supplied by the application to handle the modem input and the application's output. If the V42 module is active, use **tmModemV42DataWrite** to send data to the V42 module. If the V42 module is inactive, use **tmModemDataWrite** to send data directly to the modem. Figure 5 shows an example.

```
void ModemDataTransmitHandler(void){
  Byte Buffer[TX_APP_BUFF_SIZE];
  static Byte data[100];
  static Int message = 0;
  static Int Count = 0;
  Int numOfBytes;
  Int numOfBytesGet;

  if( v42ModuleActive ){
    if( sendV42DataToRemote ){
      numOfBytes = tmModemV42DataWritePending();
      numOfBytesGet = tmModemV42GetCharFromTxBuffer(numOfBytes, Buffer);
      tmModemV42DataWrite(numOfBytesGet, Buffer);
    }
  }else{
    if( (Count++ != MODEM_V42_DTE_BLOCK_SIZE) && (message==0) ) return;
    Count = 0;

    numOfBytes = TEXT_BLOCK_SIZE;

    numOfBytesGet = tmModemV42GetCharFromTxBuffer(numOfBytes,Buffer);
    if( numOfBytesGet == TEXT_BLOCK_SIZE ){
      if( message==0 ){
        addStartStopBits( TEXT_BLOCK_SIZE, Buffer, data );
        message = 2*TEXT_BLOCK_SIZE*8;
      }
      if( tmModemDataWritePending() > message ){
        tmModemDataWrite( message, data );
        message = 0;
      }
    }
  }
}
```

Figure 5 Code example for ModemDataTransmitHandler

Status Handlers

The modem should handle status by using handlers. Two status handlers are used: **modemLocalStatusHandler** and **V42StatusHandler**.

Name	Page
modemLocalStatusHandler	91
V42LocalStatusHandler	91

modemLocalStatusHandler

```
void ModemLocalStatusHandler(
   tmModemStatus_t  modemStatus
);
```

Parameters

`modemStatus`	Status of the modem.

Description

The application should supply a ModemLocalStatusHandler function to handle different modem conditions. In particular, if the V42Module is active, it must be properly initialized at state tmModemConnected. See the application example exModem.c on page 92.

V42LocalStatusHandler

```
void V42LocalStatusHandler(
   tmModemV42Status_t  v42Status
);
```

Parameters

`v42Status`	Status of V42 module.

Description

The application should create a **V42LocalStatusHandler** to handle different V42 conditions. See the application example exModem.c on page 92.

Modem Example: exModem.c

A modem example is included in this release. It is available in the file exModem.c. The root task in exModem.c executes the sequence of function calls described in *TriMedia V34 Modem API Overview*.

```
void root(void){
  UInt32 targs[4] = {0, 0, 0, 0};
  ptmModemCapabilities_t cap;
  tmModemInstanceSetup_t mdmSetup = {
    ModemDataTransmitHandler,        /* Name of Transmit Handler    */
    ModemDataReceiveHandler,         /* Name of Receive Handler     */
    ModemLocalStatusHandler,         /* Name of Local Status Handler*/
    V42LocalStatusHandler,           /* Name of V42 Status Handler  */
    True                             /* True if V42 is active       */
  };
  mprintf("Start V.34 modem program.\n");
  tmModemAfeInit();

/* Start the pSOS system timer.*/
  de_init( DEV_TIMER, 0, &ioretval, &dummy );

  tmModemGetCapabilities(&cap);

/* Stores in the variable v42ModuleActive the status of V42 module*/
  v42ModuleActive = mdmSetup.V42ModuleActive;
  if(tmModemOpen(&instance) != TMLIBAPP_OK){
    mprintf("Error opening modem Instance"); exit(0);
  }
  targs[0] = (UInt32)instance;
  if(tmModemInstanceSetup(instance, &mdmSetup) != TMLIBAPP_OK){
    mprintf("Invalid Instance"); exit(0);
  }
  tmModemStart(instance);

/* Start keyboard task */
  if( (err_code = t_create("PHO2", KEYBOARD_PRIO, 40960, 40960, 0,
    &key_task)) != PSOS_OK )
    mprintf( "Can’t create keyboard_task (err = 0x%x).\n", err_code );
  if( (err_code = t_start(key_task, T_PREEMPT | T_TSLICE | T_ISR,
    KeyboardTask, targs)) != PSOS_OK )
    mprintf( "Can’t start keyboard task (err = 0x%x).\n", err_code );
  t_suspend(0L);
}
```

The code above shows that, first, the AFE is initialized by a call to **tmModemAfeInit**. It obtains the modem capabilities structure by calling **tmModemGetCapabilities**. It then calls **tmModemOpen** to obtain a modem instance, which is passed to **tmModemInstance-Setup**. **tmModemInstanceSetup** uses a variable with the instance settings (**mdmSetup**), chosen in the definition of **mdmSetup**. **mdmSetup** contains the names of the transmit, receive and status handlers, and also determines whether the V42 module is active. The modem is started by invoking **tmModemStart**. After the modem is started, a “keyboard task” is started. The purpose of the keyboard task is to obtain V25Ter commands from the console, and invoke **tmModemV25SendCommand**.

The application tasks are created in the **tmModemConnected** status of the **ModemLocal-StatusHandler**. The function **creatTxRxTasks** creates the receive and transmit tasks.

The receive task is named **DataRxTask**. It receives data from the V42, if the V42 module is active, by invoking **tmModemV42GetCharFromRxBuffer**. If V42 is not active, it obtains data directly from the datapump using the variable **dest**.

The transmit task is named **DataTxTask.** It sends a string “TriMedia softmodem” or “THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG’S BACK” to the V42 module by invoking **tmModemV42PutCharInTxBuffer**.

The receive handler is named **ModemDataReceiveHandler**. If V42 is not active, it gets data from the variable data. If V42 is active, it invokes **tmModemV42PutCharInRxBuffer** to transfer memory locations pointed by data to the V42 buffer. If V42 is active, it sends characters to the screen by invoking **mprintf**.

The transmit handler is named **ModemDataTransmitHandler**. It invokes **tmModemV42-GetCharFromTxBuffer** to transmit data directly to the datapump or to the V42 module in both cases. **tmModemV42DataWritePending** verifies the number of entries that are available in the buffer before transmission.

If the V42 module is active, some commands must be issued in the **ModemLocalStatus-Handler** and the **V42LocalStatusHandler**. See *Interfacing V42 with the Modem* on page 59.

PPP Example: exModemPPP.c

Point-to-Point protocol (PPP) is a data link layer protocol that uses multiple network layer packets to operate over a serial connection. PPP defines a protocol for link control, various network control protocols (NCP), and authetication protocols. The TM V34 Modem uses pSOS/pNA+ to establish a PPP connection. Figure 6 shows the overall architecture of the PPP protocol stack using pSOS.

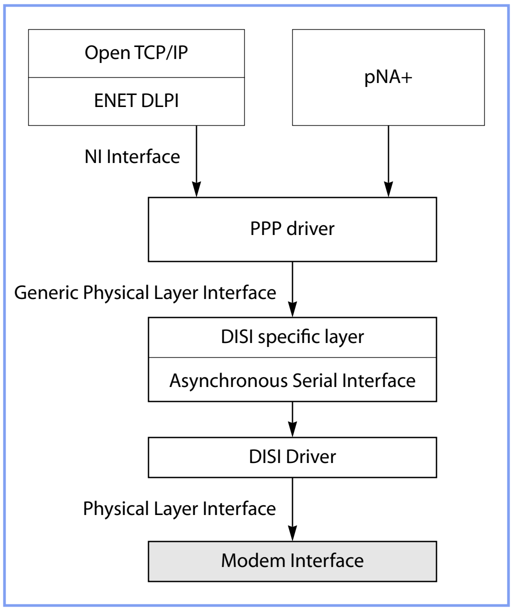


Figure 6 pNA+ PPP Protocol Stack

Figure 6 shows that the interface between the modem and PPP is performed by the Device Independent Serial Interface (DISI) driver and the DISI specific layer. The DISI specific layer provides a generic interface to the PPP driver, but specifically deals with the DISI driver. The DISI specific layer communicates with the DISI driver through an asynchronous serial interface, which is suitable for low bit rates from V34 modems. Refer to pSOS/pNA+ documentation for additional information.

exModemPPP.c is the name of the file containing the example of integration of modem and PPP. In the root task, a sequence of calls starts the modem, as described in the exModem.c example. The main difference here is the naming of the I/O handlers: in the **mdmSetup** variable containining the modem setup structure, the receive handler is **TcpipModemDataReceiveHandler** and the transmit handler is **TcpipModemDataTransmitHandler**. Also, in the “keyboard task,” once it is verified that the command string contains a dialing command (**ATDT**), the **tcpip** function is invoked. The **tcpip** function is defined in the file client.c.

The I/O Handlers are defined in the file serial.c. They use the same functions described in the exModem.c example for transferring data to/from the V42 module: **tmModemV42-PutCharInRxBuffer** and **tmModemV42GetCharFromRxBuffer**. These functions transmit and receive data from the TCP/IP connection to/from the V42 module. Refer to pNA documentation for additional information.

The example in exModemPPP.c uses several other files for the PPP implementation, described below:

- diti.c The "Device-Independent Terminal Interface," as supplied by ISI. This file converts the pSOS device interface (**de_write**, etc) into the "Device-Independent Serial Interface" (**SerialSend,** etc) in disi.c.
- pppconf.c Defines the PPP configuration, including serial and modem configurations, the hostname, the error callout and the PAP and CHAP secrets information. Other drivers can be hooked in by modifying this file.
- serial.c Implements a thin layer between the pSOS device interface and the set of functions used by the main application.
- trace.h Allows simple tracing to be enabled by compiling with **–DTRACING=1**.
- client.c File that contains main PPP code. The phone number for a PPP server, user name, password, and command for invoking PPP should be properly set in this file.

Known Bugs

- Modem might fail to detect busy tone.

Chapter 4

1394 FireWire API

Topic	Page
IEEE 1394 Overview	98
1394 API Overview	99
1394 API Enumerated Types	106
1394 API Data Structures	124
1394 API Functions	156
1394 Callback Functions	201

IEEE 1394 Overview

IEEE-1394 is a high-performance serial bus standard. The serial bus enables high-speed transmissions (100 Mbps to 400 Mbps) over a low-cost bus. The standard is currently being enhanced for transmission rates of 800 Mbps, 1.6 Gbps, and 3.2 Gbps. The serial bus standard also provides hot-pluggability. Although these features are good, they prove challenging for software developers.

Other committees have defined protocols on top of the IEEE-1394 specifications. The SBP-2 protocol provides mechanisms for delivering commands, data, and status, independent of the command set or device class of the peripheral. Printer and disk drive manufacturers have adopted this generic framework. Another protocol being developed is the DPP protocol for connecting to a printer. The future DTV equipped with these protocols can be used to print photographs from a TV directly to an IEEE-1394 printer.

The 'IP over 1394' protocol enables standard TCP/IP protocols to be used on top of the 1394 specifications. A web browser can be built on top of 'IP over 1394' and can access web pages through the IEEE-1394 cable.

The AV/C protocol permits connection to 1IEEE-394 digital camcorders. These camcorders stream video data on isochronous channels which can be received by a DTV and can be streamed to other streaming components in the DTV before being displayed.

The SD-DVCR Decoder component would allow data received from existing DV camcorders to be displayed on a television.

Glossary

AV/C. Audio/Video Control, as in the AV/C Digital Interface Command Set.

BMC. Bus Management Capability.

CIP. Common Isochronous Protocol.

CM. Cycle Master.

CSC. Cycle Start Capability.

CSR. Control and Status Register of a node or unit, as defined by IEEE 1394-1995.

EUID. Extended Unique Identifier, 64 bits, as defined by the IEEE. The EUID is a concatenation of a 24-bit company ID and a 40-bit number the vendor (identified by company ID) guarantees unique for all of its products.

IEC. International Electrotechnical Commission.

IRM. Isochronous Resource Management.

ISC. Isochronous Capability.

Isochronous. The essential characteristic of a time-scale or signal such that the time intervals between consecutive data transfers have either the same duration or durations that are integral multiples of the shortest duration.

mblock. A memory block allocated by the memory utility API defined in this document.

Node. An addressable device attached to Serial Bus with at least the minimum set of control registers defined by IEEE Standard 1394-1995.

Node ID. A 16-bit number, unique within the context of an interconnected group of serial buses. The node ID identifies both the source and the destination of asynchronous data packets on the serial bus. It can identify either one single device within the addressable group of serial buses (unicast) or all devices (broadcast).

Quadlet. Four bytes of data.

SBM. Serial Bus Manager. A set of functions defined in this document that handles serial bus management.

Serial Bus. The physical interconnections and higher-level protocols for the peer-to-peer transport of serial data, as defined by IEEE Standard 1394-1995.

1394 API Overview

These are the fundamental assumptions of the 1394 API.

1. There can be multiple link controllers (for example, PDIL11) to handle 1394 communication in a DTV platform. Each link controller is called a *device*.
2. There can be multiple instances of the 1394 API referencing the same device. A particular link controller is identified by the base memory address specified in the tsa1394 instance setup.
3. The API follows TSSA guidelines. Typically, applications call **tsa1394GetCapabilities**, **tsa1394Open**, **tsa1394GetInstanceSetup**, **tsa1394InstanceSetup** and **tsa1394Close**.
4. When the first application calls **tsa1394Open**, an instance number is returned to it. The application uses this instance number to call **tsa1394InstanceSetup**.
5. In the call to **tsa1394InstanceSetup**, the application passes **BaseAddress** as well as other parameters such as
   — the number of other nodes to which this link controller is connected.
   — the number of labels associated with each node to which it is connected.
   — the base address of the bus information block of **ConfigROM**.
   — the length of **configROM**.
   — the GRF size.
   — other FIFO sizes.
   — memory allocation function pointer.
   — ‘free’ function pointer.

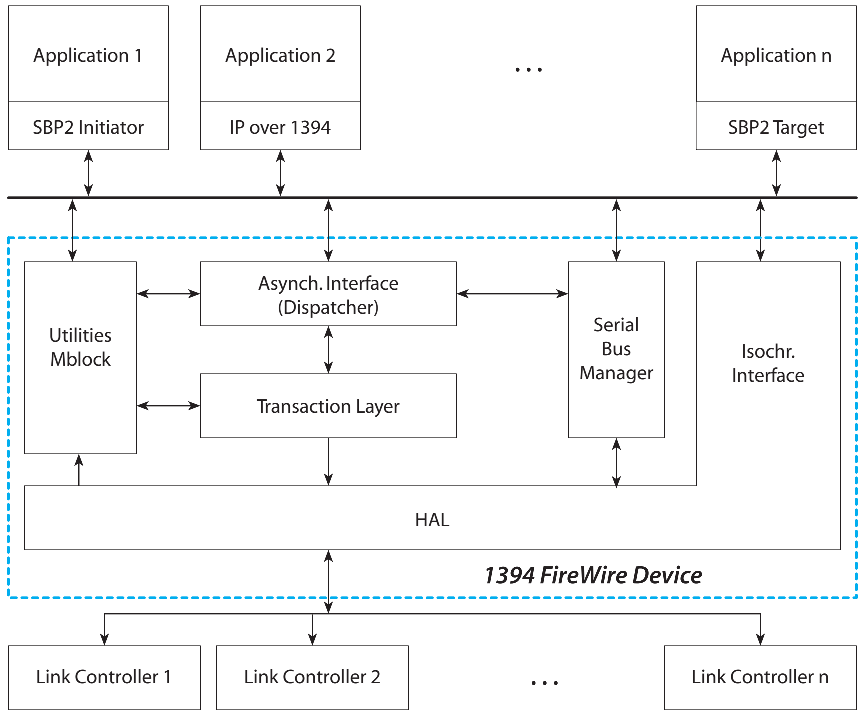
   See **tsa1394InstanceSetup** on page 161.
6. The Isochronous Resource Management (IRM) capability, Bus Management (BM) capability, Cycle Master (CM) capability and ISC (Isochronous) capability are derived

from **configROM** information, which is part of the setup structure passed to **tsa1394InstanceSetup**.

7. If a second application calls **tsa1394Open**, it will also be returned an instance number. The second application will also try to call an **tsa1394InstanceSetup**. When it does so, it passes the same base address as a parameter. If the device to which it refers is already opened by another application, indicating that this instance has been set up, the new values passed with this **tsa1394InstanceSetup** call will be ignored. However, an internal reference count indicating how many instances are tied to the link controller will be incremented.

   Each time **tsa1394Close** is called, the internal reference count decrements. When the reference count is 0, the library releases all memory used by the device (using the 'free' function specified in the tsa1394 instance setup structure).

The figure below shows the 1394 FireWire Library in a system having multiple applications that wish to use the services of the library. The 1394 FireWire Device can communicate with multiple link controllers as shown below. The figure below also shows the major blocks in the 1394 FireWire library. The main blocks in the 1394 library are the Dispatcher (for asynchronous transmission), Serial Bus Manager (SBM), Transaction Layer, and Hardware Adaptation Layer (HAL). The 1394 library also has utilities for memory pool management and for timer functionality.



Asynchronous Transmission API

The Dispatcher block allows applications to:

- Create an asynchronous channel.
- Register their Control and Status Register (CSR) address space with the 1394 library.
- Register for serial bus events (e.g., bus reset) and control events (e.g., CSR content update into physical memory). This means the application is notified of events.
- Send asynchronous requests and responses to target nodes on the bus.

The application also specifies the FireWire device to which it wants to create an asynchronous channel, through the 1394 API instance. After completing necessary operations, applications may deregister their address spaces and the serial bus and control events registered earlier. Subsequently, an application may destroy the asynchronous channel.

Isochronous Transmission API

The isochronous setup provides an API, which the application uses to set up the relevant parameters in the link controller to facilitate transmission and reception of isochronous data across the 1394 serial bus. API functions pertaining to actual data flow do not fall under the scope of this document, as they are tightly coupled with Video-In/Video-Out Pins of a Trimedia processor. An application must create a channel handle with the 1394 FireWire library. The application can then use the isochronous setup APIs for setting up the IEC-61883-specific parameters and the CIP header parameters in the link controller registers. The isochronous setup API also includes start transmission and stop transmission, start reception and stop reception functions. A function to delete the channel handle is also provided.

Serial Bus Manager API

The Serial Bus Manager (SBM) does node discovery and bus enumeration on every bus reset. It reads the configuration ROM of each node and maintains information such as maximum record size and EUID corresponding to each node. It also records the maximum speed between different nodes on the bus.

Applications can access this information maintained in the 1394 library. Applications specify the tsa1394 instance to obtain the information regarding a particular instance of FireWire device. Applications may also invoke control requests that force a reset or force a particular node to make its link layer active. Applications may also query the SBM to obtain information regarding, for example, topology and speed maps.

Transaction Layer

The transaction layer does not provide any services for applications. Other blocks in the 1394 library use the services of the transaction layer. The transaction layer provides services for split transaction management and transaction label management. The services provided by the transaction layer are used by the other blocks of the 1394 software library and are not directly available to applications. A retry mechanism is not provided by the transaction layer, but is instead directly supported by the link controllers.

Memory Utility API

This utility provides services for managing a memory pool. The purpose of memory block management is to avoid copying memory when data is passed between different layers in the system, thus providing efficient throughput. When different layers in a hierarchy want to add headers required by their protocol, they can add the required information in new memory blocks and link them with the original blocks. Applications will benefit from the memory management API of the 1394 library. It creates a pool of memory blocks of different sizes. Applications may request the allocation of memory blocks, which they can use for transferring data to other nodes on the system. Once data is sent by the HAL, it frees the blocks to be reused by the system. Applications may also use their own memory. The pool also recognizes the concept of messages. A message consists of one or more data blocks. Data blocks in a message are linked and messages themselves are also linked.

API Usage

This section describes the way an application may use the services of the 1394 library.

TSA Generic API

When an application wants to use the 1394 FireWire Library services, it will typically get the capabilities of the FireWire library using **tsa1394GetCapabilities**. It then requests an instance of a FireWire device using **tsa1394Open**. **tsa1394Open** creates an instance for the application, and return a unique instance number to the application. This instance number is to be used by the application in all subsequent calls to the 1394 FireWire library.

A call to **tsa1394GetInstanceSetup** returns the default instance setup information, if the device is not set up, or the current instance setup, if the device is already setup. **tsa1394InstanceSetup**, when called for the first time, initializes that particular FireWire device and allocates memory using the application-provided ‘alloc’ function. In all other cases, **tsa1394InstanceSetup** increases the reference count of the FireWire device that uses it. It installs the interrupt handler and make the 1394 FireWire device active. By calling **tsa1394Close**, an application informs the FireWire library that it is no longer interested in using the device. When the last call to **tsa1394close** is made, (i.e., the reference count to the device is 0) **tsa1394close** then releases all resources allocated to all 1394 FireWire devices.

tsa1394Close calls the application's 'free' routine to release the memory used by the device. The FireWire device will remain open until all applications have called **tsa1394Close**. Each call to **tsa1394Close** decreases the reference count for the device. The application should never attempt to free the memory allocated using the 'alloc' call.

Asynchronous Transmission API

Whenever an application wants to use the services of the 1394 library, it determines whether an instance of the FireWire device exists. Using this instance, it creates a context in the library by calling **tsa1394AsyncCreateAsyncChHandle**, which returns an asynchronous channel to be used in subsequent calls to the library. The application then registers its address space, serial bus management events and control events, by calling **tsa1394AsyncRegisterAddressSpace**, **tsa1394AsyncRegisterSbEvent**, and **tsa1394AsyncRegisterControlEvent**. The address space can be registered for read, write, and lock operations. For an example of usage of these functions, see *1394 API Functions*.

Subsequently, the application calls **tsa1394AsyncSendRequest** to send read or write requests to any other node using asynchronous mechanisms. Applications can register callback functions while sending read, write, and lock requests. If the application does register a callback function, the 1394 library calls this callback function, when it receives corresponding read, write, or lock confirmation for the request sent. For an example of **tsa1394AsyncSendRequest**, where data must be written to a destination node after allocating memory from a block (**tsa1394AllocbFromPool**), see *1394 API Functions*.

Subsequently, upon completion of all required transactions, the application deregisters address space, serial bus events and control events, and finally destroys the asynchronous channel itself. It calls **tsa1394AsyncDeregisterAddressSpace**, **tsa1394AsyncDeregisterSbEvent**, **tsa1394AsyncDeregisterControlEvent**, and **tsa1394AsyncDestroyChannelHandle** for these purposes.

The application may destroy its context. If it has not done so, the 1394 library detects read and writes to the registered address space. When it detects these, it calls the registered callback functions. Also, whenever serial bus events such as bus reset, bus resets complete, etc., occur, their corresponding callback functions are called.

When an application wants to send a read, write or lock transaction to a destination node, it calls **tsa1394AsyncSendRequest**. This can optionally specify a callback function to be called when a confirmation to this request is received.

Isochronous Transmission API

An application uses Isochronous APIs to send or receive Isochronous data. To send Isochronous data, the application would use **tsa1394IsochCreateChannelHandle** to allocate a channel handle. Subsequently the application will use **tsa1394IsochSetupChannel** for setting the IEC61883 Tx packing information and the CIP header information. The application would then call **tsa1394IsochStart** for starting the transmission. If the application wishes to stop isochronous data transmit, it would make a call **tsa1394IsochStop** and subsequently to start it would use, **tsa1394IsochStart**. After completion, applications can call **tsa1394IsochDestroyChannelHandle**.

To receive Isochronous data, the application would use **tsa1394IsochCreateChHandle** to allocate a channel handle. Subsequently, the application will use **tsa1394IsochSetupChannel** for setting the IEC61883 Rx unpacking information and the CIP header information. The application would then call **tsa1394IsochStart** for starting the reception. If the application wishes to stop isochronous data receive, it would make a call **tsa1394IsochStop** and subsequently to start it would use, **tsa1394IsochStart**. After completion, applications can call **tsa1394IsochDestroyChannelHandle**.

Serial Bus Management API

At any time during its operation, an application may use the Serial Bus manager (SBM) APIs to get information about the state of the bus or node and the application may request certain control actions like initiating bus reset, or initializing the transaction layer in the node. For initiating the above specified control actions, **tsa1394SbmCntrlReq** must be used with appropriate parameters passed. Due to bus reset, the node ID of a destination for data transfer may have changed. The application may wish to confirm the node ID for a given EUID and may use the services of the SBM to get this information. This uses **tsa1394SbmGetNodeIdForEuid**. All these functionalities are made available through the SBM APIs listed above.

When data has to be transferred to another node, the maximum speed between these two nodes may be obtained from the SBM. This information may be used by the application while initiating a data transfer. If there are problems encountered during the data transfer, the application may choose to try the data transfer again with a lower speed. **tsa1394SbmGetSpeed2Node** returns the speed between the current node and a destination node. Though a particular speed may allow a record size, a destination node may specify a different record size. This is stored in the BusInfoBlock of the configuration ROM. An application may use **tsa1394SbmGetMaxRec** to get this information

Memory Management API

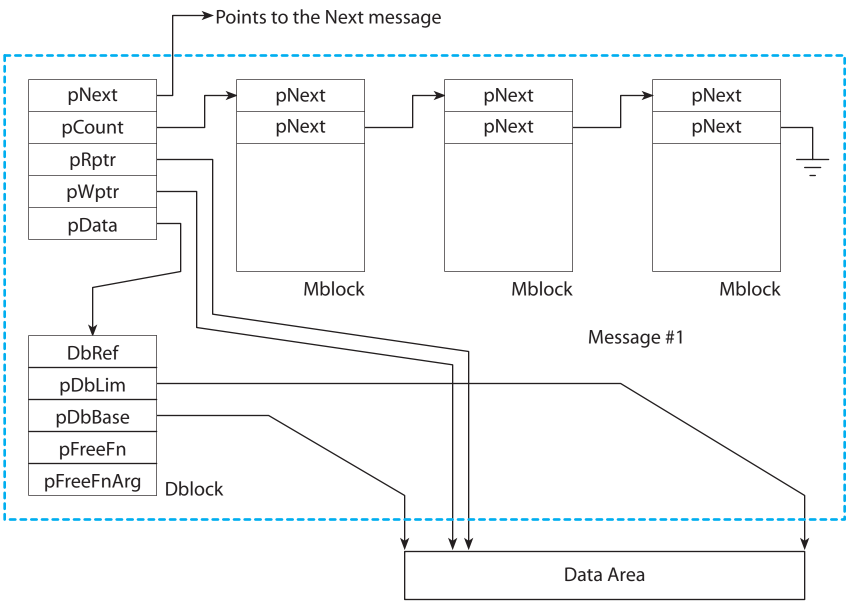
There are some utilities provided for managing memory. This is provided by the mblock utility. During initialization, a memory pool is created to receive packets from remote nodes. This is done by internals in the library. There can be a default memory pool created in a system with multiple applications. Applications can separately create their own memory pools. A memory pool consists of multiple blocks of different sizes. For example, a memory pool can be created with 5 blocks of size 512, 20 blocks of size 1024, etc. The application can allocate this buffer size, and pass it to **tsa1394MblkInit** for mblock pool initialization.

Whenever an application needs memory for data transfer, it can request an mblock allocation. This could be requested from a specified pool, or from a default pool which can all applications can use. When an application requests an mblock from a default pool, it need not specify the pool ID. For allocation from a specified pool, **tsa1394AllocbFromPool** should be used, else **tsa1394Allocb** may be used.

When an application tries to use the Asynchronous transmission API and resources are not available the 1394 library would return error messages to the application. The application has to register for a callback when the resources are available. This callback will be called when resources become available. If multiple applications had registered for the callback, the callbacks of all applications will be called.

The mblocks received from a pool can be used by applications to form messages. A message is a list of mblocks linked together. The pointer used to link the blocks of a message is pCont. Multiple messages can also be linked together and the pNext pointer does this.

The following is a pictorial view of this linked list of memory blocks:



An application may determine whether it uses its own memory for the data area rather than asking for allocation of blocks from the memory pool. Utilities are also provided for this function. Use **tsa1394EsbAlloc** or **tsa1394EsbAllocFromPool** as appropriate.

Utilities are also provided to applications for freeing blocks, copying blocks and duplicating blocks. Applications also have facilities to link blocks and unlink them as necessary.

Applications also need to free memory blocks earlier allocated by the HAL (hardware layer). Applications will use **tsa1394Freeb** or **tsa1394FreeMsg** as appropriate.

All these functionalities are made available through the API, presented next.

1394 API Enumerated Types

These are the enumerated types:

Name	Page
tsa1394FwError_t	107
tsa1394SbmError_t	110
tsa1394SupportMuxMode_t	112
tsa1394Speed_t	113
tsa1394AsynClbkType_t	114
tsa1394ExtTCode_t	115
tsa1394RespCode_t	116
tsa1394trType_t	117
tsa1394trReqStatus_t	118
tsa1394SbEvent_t	119
tsa1394MaxRec_t	120
tsa1394SbmCntrlOp_t	121
tsa1394AddrType_t	122
tsa1394CtrlEvent_t	123

tsa1394FwError_t

```
typedef enum _tsa1394FwError {
/* General Errors */
   P1394_GENERR_NULLARG                       = P1394_FW_GENERR_BASE,
   P1394_GENERR_INTERNAL                      = P1394_FW_GENERR_BASE+1,
   P1394_GENERR_CLIINIT                       = P1394_FW_GENERR_BASE+2,
   P1394_GENERR_INVALID_FN_PTR                = P1394_FW_GENERR_BASE+3,
/* System Errors */
   P1394_SYSERR_NOMEM                         = P1394_FW_SYSERR_BASE,
   P1394_SYSERR_NORESOURCES                   = P1394_FW_SYSERR_BASE+1,
/* Errors from the Dynamic Q manipulation lib */
   P1394_DQ_ERR_NOUNLOCKFN                    = P1394_DQ_ERR_BASE,
   P1394_DQ_ERR_NOLOCKFN                      = P1394_DQ_ERR_BASE+1,
   P1394_DQ_ERR_LOCKING                       = P1394_DQ_ERR_BASE+2,
   P1394_DQ_ERR_UNLOCKING                     = P1394_DQ_ERR_BASE+3,
   P1394_DQ_ERR_INCOMPATIBLE                  = P1394_DQ_ERR_BASE+4,
/* Errors from the Transparent Mem manipulation lib */
   P1394_TSDM_ERR_INVALID_MEMID               =   P1394_TSDM_ERR_BASE,
/* Errors from the Dispatcher module */
   P1394_DSPT_ERR_GEN_ERR                     = P1394_DSPT_ERR_BASE+0,
   P1394_DSPT_ERR_INVALID_TRANS_TYPE          = P1394_DSPT_ERR_BASE+1,
   P1394_DSPT_ERR_ADDR_NOT_REGISTERED         = P1394_DSPT_ERR_BASE+2,
   P1394_DSPT_ERR_INVALID_SBEVENT_REGISTER_ID = P1394_DSPT_ERR_BASE+3,
   P1394_DSPT_ERR_INVALID_SBEVENT             = P1394_DSPT_ERR_BASE+4,
   P1394_DSPT_ERR_INVALID_EXTENDED_TCODE      = P1394_DSPT_ERR_BASE+5,
   P1394_DSPT_ERR_NOCALLBACK                  = P1394_DSPT_ERR_BASE+6,
   P1394_DSPT_ERR_INVALID_DATA_IND            = P1394_DSPT_ERR_BASE+7,
   P1394_DSPT_ERR_INVALID_DATA_CNFM           = P1394_DSPT_ERR_BASE+8,
   P1394_DSPT_ERR_NOTREGISTERED               = P1394_DSPT_ERR_BASE+9,
   P1394_DSPT_ERR_ADDRSPACE_INUSE             = P1394_DSPT_ERR_BASE+10,
   P1394_DSPT_INVALID_FWDRV                   = P1394_DSPT_ERR_BASE+11,
   P1394_DSPT_INVALID_APPCONTEXT              = P1394_DSPT_ERR_BASE+12,
   P1394_DSPT_TOOMANY_FWIFS                   = P1394_DSPT_ERR_BASE+13,
   P1394_DSPT_ERR_DUPLICATE_SBEVEN            = P1394_DSPT_ERR_BASE+14,
   P1394_DSPT_ERR_INVALID_NOTICE              = P1394_DSPT_ERR_BASE+15,
   P1394_DSPT_ERR_DUPLICATE_NOTICE            = P1394_DSPT_ERR_BASE+16,
   P1394_DSPT_INVALID_IFNUM                   = P1394_DSPT_ERR_BASE+17,
   P1394_DSPT_TX_DISABLED                     = P1394_DSPT_ERR_BASE+18,
   P1394_DSPT_ERR_INVALID_ADDR_TYPE           = P1394_DSPT_ERR_BASE+19,
   P1394_DSPT_ERR_INVALID_FN_PTR              = P1394_DSPT_ERR_BASE+20,
   P1394_DSPT_ERR_NO_MEMORY                   = P1394_DSPT_ERR_BASE+21,
   P1394_DSPT_ERR_NULL_FREE_FN                = P1394_DSPT_ERR_BASE+22,
} tsa1394FwError_t;
```

Fields

`P1394_GENERR_NULLARG`	NULL argument.
`P1394_GENERR_INTERNAL`	Internal error.
`P1394_GENERR_CLIINIT`	General error in client initialization.
`P1394_GENERR_INVALID_FN_PTR`	Invalid function pointer.
`P1394_SYSERR_NOMEM`	No memory.
`P1394_SYSERR_NORESOURCES`	No resources.
`P1394_DQ_ERR_NOUNLOCKFN`	No unlock function.
`P1394_DQ_ERR_NOLOCKFN`	No lock function.
`P1394_DQ_ERR_LOCKING`	Locking error.
`P1394_DQ_ERR_UNLOCKING`	Unlocking error.
`P1394_DQ_ERR_INCOMPATIBLE`	Data quest incompatible.
`P1394_TSDM_ERR_INVALID_MEMID`	Invalid memory ID.
`P1394_DSPT_ERR_GEN_ERR`	General error.
`P1394_DSPT_ERR_INVALID_TRANS_TYPE`	Invalid transmition type.
`P1394_DSPT_ERR_ADDR_NOT_REGISTERED`	Address not registered.
`P1394_DSPT_ERR_INVALID_SBEVENT_REGISTER_ID`	Invalid serial bus event register ID.
`P1394_DSPT_ERR_INVALID_SBEVENT`	Invalid serial bus event.
`P1394_DSPT_ERR_INVALID_EXTENDED_TCODE`	Invalid extended transaction code.
`P1394_DSPT_ERR_NOCALLBACK`	No callback function provided.
`P1394_DSPT_ERR_INVALID_DATA_IND`	Invalid data Indication.
`P1394_DSPT_ERR_INVALID_DATA_CNFM`	Invalid data confirmation.
`P1394_DSPT_ERR_NOTREGISTERED`	Address/functions not registered.
`P1394_DSPT_ERR_ADDRSPACE_INUSE`	Address space already in use.
`P1394_DSPT_INVALID_FWDRV`	Invalid device.
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel.
`P1394_DSPT_TOOMANY_FWIFS`	Too many interfaces.
`P1394_DSPT_ERR_DUPLICATE_SBEVENT`	Duplicated serial bus event.
`P1394_DSPT_ERR_INVALID_NOTICE`	Invalid notice.
`P1394_DSPT_ERR_DUPLICATE_NOTICE`	Duplicated notice.
`P1394_DSPT_INVALID_IFNUM`	Invalid interface number.
`P1394_DSPT_TX_DISABLED`	Transmission disabled.

`P1394_DSPT_ERR_INVALID_ADDR_TYPE`	Invalid address type.
`P1394_DSPT_ERR_INVALID_FN_PTR`	Invalid function pointer.
`P1394_DSPT_ERR_NO_MEMORY`	No memory is available.
`P1394_DSPT_ERR_NULL_FREE_FN`	Null memory free function.

Description

Enumerates the 1394 FireWire errors.

tsa1394SbmError_t

```
typedef enum _tsa1394SbmError {
   P1394_SBM_ERR_UNEXPECTED_CSROP   = P1394_SBM_ERR_BASE,
   P1394_SBM_ERR_INVALID_CSRO       = P1394_SBM_ERR_BASE+ 1,
   P1394_SBM_ERR_NON_QUAD_CSROP     = P1394_SBM_ERR_BASE+ 2,
   P1394_SBM_TOOMANY_FWIFS          = P1394_SBM_ERR_BASE+ 3,
   P1394_SBM_ERR_IN_RESET           = P1394_SBM_ERR_BASE+ 4,
   P1394_SBM_ERR_IF_REGISTERED      = P1394_SBM_ERR_BASE+ 5,
   P1394_SBM_ERR_IF_NOTREGISTERED   = P1394_SBM_ERR_BASE+ 6,
   P1394_SBM_INVALID_FWDRV          = P1394_SBM_ERR_BASE+ 7,
   P1394_SBM_SPEED_UNKNOWN          = P1394_SBM_ERR_BASE+ 8,
   P1394_SBM_EUID_UNKNOWN           = P1394_SBM_ERR_BASE+ 9,
   P1394_SBM_MAXREC_UNSPECIFIED     = P1394_SBM_ERR_BASE+10,
   P1394_SBM_MAXREC_RESERVED        = P1394_SBM_ERR_BASE+11,
   P1394_SBM_MAXREC_UNKNOWN         = P1394_SBM_ERR_BASE+12,
   P1394_SBM_INVALID_CNTRLOP        = P1394_SBM_ERR_BASE+13,
   P1394_SBM_TOPOLOGY_NOT_BUILT     = P1394_SBM_ERR_BASE+14,
   P1394_SBM_NO_IRM_ON_BUS          = P1394_SBM_ERR_BASE+15,
   P1394_SBM_INVALID_PHYS_ID        = P1394_SBM_ERR_BASE+16,
   P1394_SBM_CLBITS_UNKNOWN         = P1394_SBM_ERR_BASE+17,
   P1394_SBM_COMPARE_SWAP_FAILED    = P1394_SBM_ERR_BASE+18,
   P1394_SBM_ERR_NO_MEMORY          = P1394_SBM_ERR_BASE+19,
   P1394_SBM_ERR_INVALID_FN_PTR     = P1394_SBM_ERR_BASE+20,
   P1394_SBM_ERR_INVALID_FREE_FN    = P1394_SBM_ERR_BASE+21,
   P1394_SBMBUF_ERR_INVALID_FREE_FN = P1394_SBM_ERR_BASE+22,
   P1394_SBMBUF_ERR_INVALID_FN_PTR  = P1394_SBM_ERR_BASE+23,
   P1394_SBMBUF_ERR_NO_MEMORY       = P1394_SBM_ERR_BASE+24
} tsa1394SbmError_t;
```

Fields

`P1394_SBM_ERR_UNEXPECTED_CSROP`	Unexpected read, write, or lock CSR operation.
`P1394_SBM_ERR_INVALID_CSROP`	Invalid Control and Status Register operation.
`P1394_SBM_ERR_NON_QUAD_CSROP`	Request blocked on quadlet registration.
`P1394_SBM_TOOMANY_FWIFS`	Too many FireWire instances.
`P1394_SBM_ERR_IN_RESET`	Reset error.
`P1394_SBM_ERR_IF_REGISTERED`	Registered error.
`P1394_SBM_ERR_IF_NOTREGISTERED`	Not registered.
`P1394_SBM_INVALID_FWDRV`	Invalid device.
`P1394_SBM_SPEED_UNKNOWN`	Speed unknown.
`P1394_SBM_EUID_UNKNOWN`	EUID unknown error.
`P1394_SBM_MAXREC_UNSPECIFIED`	Maxrec unspecified.
`P1394_SBM_MAXREC_RESERVED`	Maxrec reserved.
`P1394_SBM_MAXREC_UNKNOWN`	Maxrec unknown.

`P1394_SBM_INVALID_CNTRLOP`	Invalid control loop.
`P1394_SBM_TOPOLOGY_NOT_BUILT`	Topology not built error.
`P1394_SBM_NO_IRM_ON_BUS`	No IRM on bus error.
`P1394_SBM_INVALID_PHYS_ID`	Invalid physical ID.
`P1394_SBM_CLBITS_UNKNOWN`	Info enquired on a node in remote bus.
`P1394_SBM_COMPARE_SWAP_FAILED`	Compare swap failed.
`P1394_SBM_ERR_NO_MEMORY`	SBM no memory.
`P1394_SBM_ERR_INVALID_FN_PTR`	SBM invalid function pointer.
`P1394_SBM_ERR_INVALID_FREE_FN`	SBM invalid free memory function.
`P1394_SBMBUF_ERR_INVALID_FREE_FN`	SBM buffer invalid free memory function.
`P1394_SBMBUF_ERR_INVALID_FN_PTR`	SBM buffer invalid function pointer.
`P1394_SBMBUF_ERR_NO_MEMORY`	SBM buffer no memory.

Description

Enumerates the 1394 Serial Bus Management (SBM) errors.

tsa1394SupportMuxMode_t

```
typedef enum _tsa1394SupportMuxMode {
   P1394_MUX_VO_TO_1394_MODE  =  0x01,
   P1394_MUX_1394_TO_VI_MODE,
   P1394_MUX_CPU_BYPASS_MODE
} tsa1394SupportMuxMode_t;
```

Fields

`P1394_MUX_VO_TO_1394_MODE`	1394 coupled with video-out in transmitting mode.
`P1394_MUX_1394_TO_VI_MODE`	1394 coupled with video-in in receiving mode.
`P1394_MUX_CPU_BYPASS_MODE`	The data path bypasses the TM chip.

Description

Flags to indicate the configuration of the 1394 multiplexed with either video-in or video-out in receiving or transmiting operation.

tsa1394Speed_t

```
typedef enum _tsa1394CblPhySpeed {
   P1394_FWCPHY_SPEED_S100     = 0,
   P1394_FWCPHY_SPEED_S200,
   P1394_FWCPHY_SPEED_S400,
   P1394_FWCPHY_SPEED_HIGHEST  = P1394_FWCPHY_SPEED_S400,
   P1394_FWCPHY_SPEED_SANY,
   P1394_FWCPHY_SPEED_UBOUND   = P1394_FWCPHY_SPEED_SANY
} tsa1394Speed_t;
```

Fields

`P1394_FWCPHY_SPEED_S100`	Transmission speed, 100 Mbps (i.e., base speed).
`P1394_FWCPHY_SPEED_S200`	Transmission speed, 200 Mbps.
`P1394_FWCPHY_SPEED_S400`	Transmission speed, 400 Mbps.
`P1394_FWCPHY_SPEED_HIGHEST`	Transmission speed, 400 Mbps.
`P1394_FWCPHY_SPEED_SANY`	Any of S100, S200 and S400.
`P1394_FWCPHY_SPEED_UBOUND`	For internal bound checks.

Description

Enumerates 1394 speed types.

tsa1394AsynClbkType_t

```
typedef enum _tsa1394AsynClbkType {
   P1394_FW_CLBK_IND = 0,
   P1394_FW_CLBK_CNF,
   P1394_FW_CLBK_PHW,
   P1394_FW_CLBK_DES,
   P1394_FW_CLBK_RESET,
   P1394_FW_CLBK_SIDCOMP,
   P1394_FW_CLBK_ISORX,
   P1394_FW_CLBK_ISOTX
} tsa1394AsynClbkType_t;
```

Fields

P1394_FW_CLBK_IND	Callback indication.
P1394_FW_CLBK_CNF	Callback confirmation.
P1394_FW_CLBK_PHW	Callback physical write.
P1394_FW_CLBK_DES	Descriptor available; not applicable.
P1394_FW_CLBK_RESET	Control block reset type.
P1394_FW_CLBK_SIDCOMP	Self-ID completion callback function. This callback function is trickled when communication is reset or changed and after the root node has received all its children’s ID information.
P1394_FW_CLBK_ISORX	Isochronous receiving type.
P1394_FW_CLBK_ISOTX	Isochronous transmitting type.

Description

Enumerates 1394 asynchronous control block types.

tsa1394ExtTCode_t

```
typedef enum _tsa1394ExtTCode {
   P1394_FWLL_ETC_RESERVED1      = Ø,
   P1394_FWLL_ETC_MASK_SWAP,
   P1394_FWLL_ETC_COMPARE_SWAP,
   P1394_FWLL_ETC_FETCH_ADD,
   P1394_FWLL_ETC_LITTLE_ADD,
   P1394_FWLL_ETC_BOUNDED_ADD,
   P1394_FWLL_ETC_WRAP_ADD,
   P1394_FWLL_ETC_VENDOR_DEPENDENT,
   P1394_FWLL_ETC_RESERVED_2,
   P1394_FWLL_ETC_RESERVED_N     = ØxFFFF
} tsa1394ExtTCode_t;
```

Fields

`P1394_FWLL_ETC_RESERVED1`	Extended transaction code reserved1.
`P1394_FWLL_ETC_MASK_SWAP`	Extended transaction code swap.
`P1394_FWLL_ETC_COMPARE_SWAP`	Extended transaction code compare swap.
`P1394_FWLL_ETC_FETCH_ADD`	Extended transaction code fetch add.
`P1394_FWLL_ETC_LITTLE_ADD`	Extended transaction code little add.
`P1394_FWLL_ETC_BOUNDED_ADD`	Extended transaction code bounded add.
`P1394_FWLL_ETC_WRAP_ADD`	Extended transaction code swap add.
`P1394_FWLL_ETC_VENDOR_DEPENDENT`	Extended transaction code vendor-dependent.
`P1394_FWLL_ETC_RESERVED_2`	Extended transaction code reserved2.
`P1394_FWLL_ETC_RESERVED_N`	Extended transaction code reserved n.

Description

Enumerates 1394 extended transaction codes.

tsa1394RespCode_t

```
typedef enum _tsa1394RespCode {
   P1394_FWLL_RC_COMPLETE      = Ø,
   P1394_FWLL_RC_RESERVED1,
   P1394_FWLL_RC_RESERVED2,
   1394_FWLL_RC_RESERVED3,
   P1394_FWLL_RC_CONFLICT_ERR,
   P1394_FWLL_RC_DATA_ERR,
   P1394_FWLL_RC_TYPE_ERR,
   P1394_FWLL_RC_ADDRESS_ERR,
   P1394_FWLL_RC_RESERVED_4,
   P1394_FWLL_RC_RESERVED_N    = ØxØF,
   P1394_FWLL_RC_UBOUND        = P1394_FWLL_RC_RESERVED_N
} tsa1394RespCode_t;
```

Fields

`P1394_FWLL_RC_COMPLETE`	Response code complete.
`P1394_FWLL_RC_RESERVED1`	Response code reserved 1.
`P1394_FWLL_RC_RESERVED2`	Response code reserved 2.
`P1394_FWLL_RC_RESERVED3`	Response code reserved 3.
`P1394_FWLL_RC_CONFLICT_ERR`	Response code conflict error.
`P1394_FWLL_RC_DATA_ERR`	Response code data error.
`P1394_FWLL_RC_TYPE_ERR`	Response code type error.
`P1394_FWLL_RC_ADDRESS_ERR`	Response code address error.
`P1394_FWLL_RC_RESERVED_4`	Response code reserved 4.
`P1394_FWLL_RC_RESERVED_N`	Response code reserved n.
`P1394_FWLL_RC_UBOUND`	For internal software use.

Description

Enumerates 1394 response codes.

tsa1394trType_t

```
typedef enum _tsa1394trType {
   P1394_FWTR_READBLOCK   = Ø,
   P1394_FWTR_LB          = P1394_FWTR_READBLOCK,
   P1394_FWTR_READQUAD,
   P1394_FWTR_WRITEBLOCK,
   P1394_FWTR_WRITEQUAD,
   P1394_FWTR_LOCK,
   P1394_FWTR_UB          = P1394_FWTR_LOCK
} tsa1394trType_t;
```

Fields

`P1394_FWTR_READBLOCK`	Transaction type read block.
`P1394_FWTR_LB`	Lower boundary.
`P1394_FWTR_READQUAD`	Transaction type read quad.
`P1394_FWTR_WRITEBLOCK`	Transaction type write block.
`P1394_FWTR_WRITEQUAD`	Transaction type write quad.
`P1394_FWTR_LOCK`	Transaction type lock.
`P1394_FWTR_UB`	Upper boundary.

Description

Enumerates 1394 transaction types.

tsa1394trReqStatus_t

```
typedef enum _tsa1394trReqStatus{
   P1394_FWTL_RSTAT_COMPLETE    = Ø,
   P1394_FWTL_RSTAT_LB          = P1394_FWTL_RSTAT_COMPLETE,
   P1394_FWTL_RSTAT_TIMEOUT,
   P1394_FWTL_RSTAT_ACK_MISS,
   P1394_FWTL_RSTAT_RETRY_LIMIT,
   P139g4_FWTL_RSTAT_DATA_ERR,
   P1394_FWTL_RESET_EV,
   P1394_FWTL_RSTAT_UB          = P1394_FWTL_RESET_EV
} tsa1394trReqStatus_t;
```

Fields

`P1394_FWTL_RSTAT_COMPLETE`	Request status complete.
`P1394_FWTL_RSTAT_LB`	Transaction layer request status lower boundary.
`P1394_FWTL_RSTAT_TIMEOUT`	Request status timeout.
`P1394_FWTL_RSTAT_ACK_MISS`	Request status acknowledge missing.
`P1394_FWTL_RSTAT_RETRY_LIMIT`	Request status retry limit.
`P1394_FWTL_RSTAT_DATA_ERR`	Request status data error.
`P1394_FWTL_RESET_EV`	Transaction layer request status bus reset event. Not applicable on PDI1394L11.
`P1394_FWTL_RSTAT_UB`	Transaction layer request status upper boundary.

Description

Enumerates returned 1394 transaction request status codes.

tsa1394SbEvent_t

```
typedef enum _tsa1394SbEvent {
  P1394_FWSBM_EV_OCCUPANCY_VIOLATION  =  0,
  P1394_FWSBM_EV_LB = P1394_FWSBM_EV_OCCUPANCY_VIOLATION,
  P1394_FWSBM_EV_RESET_START          =  1,
  P1394_FWSBM_EV_RESET_COMPLETE       =  2,
  P1394_FWSBM_EV_CYCLE_TOO_LONG       =  3,
  P1394_FWSBM_EV_POWER_FAIL           =  4,
  P1394_FWSBM_EV_DUPLICATE_CHANNEL    =  5,
  P1394_FWSBM_EV_CRC_ERROR            =  6,
  P1394_FWSBM_EV_REQ_DATA_ERROR       =  7,
  P1394_FWSBM_EV_RESP_ACK_MISSING     =  8,
  P1394_FWSBM_EV_RESP_DATA_ERROR      =  9,
  P1394_FWSBM_EV_RESP_FORMAT_ERROR    = 10,
  P1394_FWSBM_EV_RESP_RETRY_FAILED    = 11,
  P1394_FWSBM_EV_UNEXPECTED_CHANNEL   = 12,
  P1394_FWSBM_EV_UNKNOWN_TRANS_CODE   = 13,
  P1394_FWSBM_EV_UNSOLICITED_RESPONSE = 14,
  P1394_FWSBM_EV_UB = P1394_FWSBM_EV_UNSOLICITED_RESPONSE
}  tsa1394SbEvent_t;
```

Fields

`P1394_FWSBM_EV_OCCUPANCY_VIOLATION`	SBM event occupancy violation.
`P1394_FWSBM_EV_LB`	SBM event lower boundary.
`P1394_FWSBM_EV_RESET_START`	SBM event reset start.
`P1394_FWSBM_EV_RESET_COMPLETE`	SBM event complete.
`P1394_FWSBM_EV_CYCLE_TOO_LONG`	SBM event cycle too long.
`P1394_FWSBM_EV_POWER_FAIL`	SBM event power fail.
`P1394_FWSBM_EV_DUPLICATE_CHANNEL`	SBM event duplicated channel.
`P1394_FWSBM_EV_CRC_ERROR`	SBM event CRC error.
`P1394_FWSBM_EV_REQ_DATA_ERROR`	SBM event request data error.
`P1394_FWSBM_EV_RESP_ACK_MISSING`	Response acknowledge missing.
`P1394_FWSBM_EV_RESP_DATA_ERROR`	Response data error.
`P1394_FWSBM_EV_RESP_FORMAT_ERROR`	Response format error.
`P1394_FWSBM_EV_RESP_RETRY_FAILED`	Response retry failed.
`P1394_FWSBM_EV_UNEXPECTED_CHANNEL`	Unexpected channel.
`P1394_FWSBM_EV_UNKNOWN_TRANS_CODE`	Unknown transcation code.

`P1394_FWSBM_EV_UNSOLICITED_RESPONSE`	Unsolicited response.
`P1394_FWSBM_EV_UB`	SBM event upper boundary.

Description

Enumerates 1394 SBM (Serial Bus Manager) event types.

tsa1394MaxRec_t

```
typedef enum _tsa1394MaxRec {
   P1394_FWSBM_S100_MAXREC  =  512,
   P1394_FWSBM_S200_MAXREC  = 1024,
   P1394_FWSBM_S400_MAXREC  = 2048
} tsa1394MaxRec_t;
```

Fields

`P1394_FWSBM_S100_MAXREC`	Maximum record size at 100 Mbps.
`P1394_FWSBM_S200_MAXREC`	Maximum record size at 200 Mbps.
`P1394_FWSBM_S400_MAXREC`	Maximum record size at 400 Mbps.

Description

Enumerates 1394 FireWire SBM maximum record sizes at various speeds.

tsa1394SbmCntrlOp_t

```
typedef enum _tsa1394SbmCntrlOp {
   P1394_SBM_CNTRL_RESET,
   P1394_SBM_CNTRL_INIT,
   P1394_SBM_CNTRL_LINKON,
   P1394_SBM_CNTRL_STATUS,
   P1394_SBM_CNTRL_PHYCONFIG
} tsa1394SbmCntrlOp_t;
```

Fields

`P1394_SBM_CNTRL_RESET`	SBM control reset.
`P1394_SBM_CNTRL_INIT`	SBM control initiation.
`P1394_SBM_CNTRL_LINKON`	SBM control link on.
`P1394_SBM_CNTRL_STATUS`	SBM control status.
`P1394_SBM_CNTRL_PHYCONFIG`	SBM control physical configuration.

Description

Enumerates 1394 SBM control types.

tsa1394AddrType_t

```
typedef enum _tsa1394AddrType {
   P1394_FW_ADDR_READ   = Ø,
   P1394_FW_ADDR_LB     = P1394_FW_ADDR_READ,
   P1394_FW_ADDR_WRITE,
   P1394_FW_ADDR_LOCK,
   P1394_FW_ADDR_UB     = P1394_FW_ADDR_LOCK
} tsa1394AddrType_t;
```

Fields

`P1394_FW_ADDR_READ`	Addr type read.
`P1394_FW_ADDR_LB`	Addr type lower boundary.
`P1394_FW_ADDR_WRITE`	Addr type write.
`P1394_FW_ADDR_LOCK`	Addr type lock.
`P1394_FW_ADDR_UB`	Addr type upper boundary.

Description

Enumerates 1394 address types.

tsa1394CtrlEvent_t

```
typedef enum _tsa1394CtrlEvent {
   P1394_FWCE_PHYSICAL_WRITE         = 0,
   P1394_FWCE_LB                     = P1394_FWCE_PHYSICAL_WRITE,
   P1394_FWCE_RESOURCE_AVAILABILITY  = 1,
   P1394_FWCE_COMMAND_RESET          = 2,
   P1394_FWCE_UB                     = 1394_FWCE_COMMAND_RESET
} tsa1394CtrlEvent_t;
```

Fields

`P1394_FWCE_PHYSICAL_WRITE`	Physical write.
`P1394_FWCE_LB`	Control event lower boundary.
`P1394_FWCE_RESOURCE_AVAILABILITY`	Resource availability.
`P1394_FWCE_COMMAND_RESET`	Command reset.
`P1394_FWCE_UB`	Control event upper boundary.

Description

Enumerates 1394 control event types.

1394 API Data Structures

These are the data structures:

Name	Page
tsa1394Capabilities_t	126
tsa1394Setup_t	128
tsa1394EUId_t	130
tsa1394DestOffset_t	130
tsa1394FreeRtn_t	131
tsa1394Tdatab_t	132
tsa1394MBlock_t	133
tsa1394MblkBufConfig_t	134
tsa1394DblkLink_t	135
tsa1394BusTime_t	136
tsa1394trDataCnfm_t	137
tsa1394trDataInd_t	138
tsa1394IecTxInfo_t	140
tsa1394IecRxUnpackInfo_t	141
tsa1394IsochHdrInfo_t	142
tsa1394SbResetEventInfo_t	143
tsa1394SbEventInfo_t	144
tsa1394SbmCntrlResetInitParams_t	144
tsa1394SbmCntrlPhyConfigParams_t	145
tsa1394SbmStatusInfo_t	146
tsa1394PhysWriteInfo_t	147
tsa1394CtrlEventInfo_t	147
tsa1394RdBlockReq_t	148
tsa1394RdQuadReq_t	149
tsa1394WrBlockReq_t	150
tsa1394WrQuadReq_t	151

Name	Page
tsa1394RdBlkResp_t	152
tsa1394RdQuadResp_t	153
tsa1394WrResp_t	154
tsa1394LockResp_t	155

tsa1394Capabilities_t

```
typedef struct _tsa1394Capabilities {
   tmVersion_t   versionNum;
   UInt32        ulNumSupportedInstances;
   UInt32        ulNumCurrentInstances;
   UInt32        ulNumtsa1394MaxNodesConnected;
   UInt32        ulNumtsa1394Labels;
   UInt32        ul1394Capability;
} tsa1394Capabilities_t, *ptsa1394Capabilities_t;
```

Fields

`ulNumSupportedInstances`	Maximum number of tsa1394 instances.
`ulNumCurrentInstances`	Number of current instances.
`ulNumtsa1394MaxNodesConnected`	Maximum number of nodes connected.
`UlNumtsa1394Labels`	Maximum number of rransaction labels. This indicates the maximum number of asynchronous transactions that can coexist.
`ul1394Capability`	The field can be bitwise OR'd by the following masks: **ISOC_PORT**. There are link controllers which need not have a isoch port which can be directly connected to a port on the TriMedia such as VI or VO. In this case, the library would need to read isochronous data also through the general-purpose inputs. This is currently not supported for the PDI1394L11. **AVC_HDR**. The capability to add CIP headers in the software library. **MPEG_CIP**. Indicates whether the Link controller can add MPEG headers. **CHAN_AVL**. Indicates whether the CHANNEL_AVAILABLE register is implemented in the link controller itself. **BWTH_AVL**. Indicates whether the BANDWIDTH_AVAILABLE register is implemented in the link controller itself. **PLUG_CTRL**. Indicates whether the PLUG_CONTROL register is implemented in the link controller. **IRM_CAP**. A capability available in the tsa1394 library because isochronous functionality is necessary for most applications.

BMC_CAP. A capability available in the tsa1394 library because isochronous functionality is necessary for most applications.

CMC_CAP. A capability available in the tsa1394 library because isochronous functionality is necessary for most applications.

Description

Capabilities of the 1394 library.

tsa1394Setup_t

```
typedef struct _tsa1394Setup {
   tsa1394Drv_t         ulDrvId;
   UInt32               ulBaseAddr;
   UInt32               ulIrq;
   UInt32               ulBusId;
   UInt32               ulNumNodesConnected;
   UInt32               ulNumTLabels;
   Byte                *pCfgRomData;
   UInt32               ulCfgRomDataLen;
   tsa1394AllocCbFp_t   cacheAllocFp;
   tsa1394FreeCbFp_t    cacheFreeFp;
   void                *cFreeUserData;
   tsa1394AllocCbFp_t   nCacheAllocFp;
   tsa1394FreeCbFp_t    nCacheFreeFp;
   void                *nCFreeUserData;
   UInt32               ulNumGruBufCount;
   void                *pSpecific;
   tsaErrorFunc_t       errorFunc;
   tsaProgressFunc_t    progressFunc;
} tsa1394Setup_t, *ptsa1394Setup_t;
```

Fields

`ulDrvId`	The distinguishing Driver ID.
`ulBaseAddr`	Where the controller is mapped in address space.
`ulIrq`	The controller interrupt.
`ulBusId`	The 1394 Bus ID.
`ulNumNodesConnected`	Max number of 1394 Nodes on the bus.
`ulNumTLabels`	Simultaneous maximum number of T Labels to a Node.
`pCfgRomData`	Configuration ROM data.
`ulCfgRomDataLen`	Length of the configuration ROM.
`cacheAllocFp`	Allocation function for cached memory.
`cacheFreeFp`	Free Function for cached memory.
`cFreeUserData`	User data passed to cached free.
`nCacheAllocFp`	Non-cached memory allocation function.
`nCacheFreeFp`	Non-cached memory free function for cached memory.
`cFreeUserData`	User data passed to cached free.
`nCacheAllocFp`	Non-cached memory Allocation function.
`nCFreeUserData`	User data passed to non-cached free.
`ulNumGruBufCount`	Determines general receive buffers.

`pSpecific`	Any other specific parameters.
`errorFunc`	Error callback function.
`progressFunc`	Progress function pointer.

Description

The pCfgRomData is a valid pointer to the Config ROM bus information, which specifies the Bus Management Capability (BMC), Cycle Start Capability (CSC), Isochronous Resource Management Capability (IRM) and Isochronous capability. The total length of the Configuration ROM is given by **ulCfgRomDataLen**.

The **cacheAllocFp** will be called to allocate a cache-enabled memory pool . The **nCacheAllocFp** will be called to allocate a non-cached memory pool. The **cacheFreeFp** will be called to free a cache-enabled memory pool and **nCacheFreeFp** will be called to release a non-cached memory pool. For all memory needed by the device, memory would be allocated from non cache area. The memory for other layers of FireWire would be allocated from cache area. **nCFreeUserData**/**cFreeUserData** are optional application specific parameters, which can be passed as parameters when calling **nCacheFreeFp**/**cacheFreeFp** callback functions.

The **ulDrvId** is set by the library. The library generates a unique **ulDrvId** for every device that gets initialized. **ulBaseAddr** refers to the address where the device is mapped, **ulIrq** is the controller irq and **ulBusId** is the 1394 BusId. The application can obtain **ulBaseAddr**, ulIrq by making a call to **tsa1394GetInstanceSetup**. ulNumNodesConnected refers to the max no of 1394 nodes connected, **ulNumTLabels** refers to the number of synchronous connections at any time and **ulNumGruBufCount** refers to the number of general receive buffers. Any other specific parameters can be passed as ***pSpecific**.

Note
The application should never release memory which is allocated through this alloc call for the entire run of the system.

tsa1394EUId_t

```
typedef struct _tsa1394EUId {
   UInt32   ulQuadLo;
   UInt32   ulQuadHi;
} tsa1394EUId_t, *ptsa1394EUId_t;
```

Fields

`ulQuadLo`	The lower quadlet of the EUID.
`ulQuadHi`	The higher quadlet of the EUID.

Description

The data structure defines the Extended Unique Identifier (EUID), 64 bits, as defined by the IEEE. The EUID is a concatenation of the 24-bit company_ID and a 40-bit number that the vender (identified by company_ID) guarantees unique for all its products.

tsa1394DestOffset_t

```
typedef struct _tsa1394DestOffset {
   UInt32   ulLow;
   UInt16   usHi;
} tsa1394DestOffset_t, *ptsa1394DestOffset_t;
```

Fields

`ulLow`	32-bit low portion.
`usHi`	16-bit high portion.

Description

The data structure represents 48-bit 1394 address.

tsa1394FreeRtn_t

```
typedef struct _tsa1394FreeRtn {
   void   (*FreeFunc)(void*);
   void   *FreeArg;
} tsa1394FreeRtn_t, *ptsa1394FreeRtn_t;
```

Fields

`FreeFunc`	Application provided ‘free’ function.
`FreeArg`	Pointer to data to be freed.

Description

The data structure identifies the free function and points to data to be freed.

tsa1394Tdatab_t

```
typedef struct _tsa1394Tdatab {
   long                lPoolId;
   tsa1394Tdatab_t    *pDbFreep;
   Byte               *pDbBase;
   Byte               *pDbLim;
   Byte                ucDbRef;
   Byte                ucDbType;
   tsa1394FreeRtn_t    DbFrtn;
   tsa1394MBlock_t    *pParent;
} tsa1394Tdatab_t, *ptsa1394Tdatab_t;
```

Fields

`lPoolId`	Pool ID to which data buffer belongs.
`pDbFreep`	Field used internally.
`pDbBase`	First byte of buffer.
`pDbLim`	Last byte+1 of buffer.
`ucDbRef`	Count of messages pointing to this block.
`ucDbType`	Message type.
`DbFrtn`	Pointer to function to free data buffer.
`pParent`	Pointer to the Parent’s memory block.

Description

Data block type definition.

tsa1394MBlock_t

```
typedef struct _tsa1394MBlock {
   long                   lPoolId;
   struct _tsa1394MBlock  *pNext;
   struct _tsa1394MBlock  *pCont;
   Byte                   *pRptr;
   Byte                   *pWptr;
   struct _tsa1394Tdatab  *pData;
   UInt32                  ulRequestedCount;
   UInt32                  ulUserArg;
   void                   *allocatorInfo;
} tsa1394MBlock_t, *ptsa1394MBlock_t;
```

Fields

`lPoolId`	The ID of the pool to which mblk belongs.
`pNext`	Next message on queue.
`pPrev`	Previous message on queue.
`pCont`	Next message block of message.
`pRptr`	First unread data byte in buffer.
`pWptr`	First unwritten data byte in buffer.
`pData`	Data block.
`ulUserArg`	Reserved for internal use.
`ulRequestedCount`	Reserved for internal use.
`allocatorInfo`	Allocator Info:Internal field, optional parameter.

Description

The data structure is used in memory block utility functions.

tsa1394MblkBufConfig_t

```
typedef struct _tsa1394MblkBufConfig {
   UInt32   ulBuffers;
   UInt32   ulBufSize;
} tsa1394MblkBufConfig_t, *ptsa1394MblkBufConfig_t;
```

Fields

`ulBuffers`	Number of buffers.
`ulBufSize`	Buffer size.

Description

The last entry in the array of **tsa1394MblkBufConfig** is essentially {0, 0}.

Each element in the array contains information on the number of buffers (ulBuffers) required by the application.

tsa1394DblkLink_t

```
typedef struct _tsa1394DblkLink {
   struct DblkLink   *pNext;
   long               lTwait;
   tsa1394Tdatab_t   *pDblk;
   long               lSize;
   long               lDblks;
   long               lFree;
   long               lWait;
   long               lDrops;
} tsa1394DblkLink_t, *ptsa1394DblkLink_t;
```

Fields

`pNext`	Next data block link.
`lTwait`	Wait flag for the data block.
`pDblk`	Pointer to the data block.
`lSize`	Buffer size.
`lDblks`	Number of data blocks.
`lFree`	Number of free data blocks.
`lWait`	Number of tasks waited for data block.
`lDrops`	Number of times failed to get data blk.

Description

This structure defines the Data Block Pool Header Link information.

tsa1394BusTime_t

```
typedef struct _tsa1394BusTime {
   UInt32   ulMode;
   UInt32   ulValue;
} tsa1394BusTime_t, *ptsa1394BusTime_t;
```

Fields

`ulMode`	Specify the mode to get cycle time.
`ulValue`	Specify the value used in cycle time calculation.

Description

This structure defines the mode and value to be used in calculating cycle time.

tsa1394trDataCnfm_t

```
typedef struct _tsa1394trDataCnfm{
   tsa1394trReqStatus_t   eTrStatus;
   tsa1394RespCode_t      eRespCode;
   void                  *pData;
   UInt16                 usDataLen;
} tsa1394trDataCnfm_t, *ptsa1394trDataCnfm_t;
```

Fields

eTrStatus	Contains the transaction status.
eRespCode	Contains response code which is filled by the Remote node.
pData	Contains the data itself
usDataLen	Contains the length of data.

Description

The **tsa1394trDataCnfm_t** is passed to the callback fuction specified in **tsa1394Async-SendRequest**. **eTrStatus** takes following values.

```
typedef enum {
   P1394_FWTL_RSTAT_COMPLETE,
   P1394_FWTL_RSTAT_TIMEOUT,
   P1394_FWTL_RSTAT_ACK_MISS,
   P1394_FWTL_RSTAT_DATA_ERR
} tsa1394trReqStatus_t;
```

Other fields in **tsa1394trDataCnfm_t** are valid only if **eTrStatus** is **P1394_FWTL_RSTAT_COMPLETE**. **P1394_FWTL_RSTAT_TIMEOUT** indicates there is no response to the request sent. **P1394_FWTL_RSTAT_ACK_MISS** and **P1394_FWTL_RSTAT_DATA_ERR** indicate that problem was encountered while sending a request to the remote node. **eRespCode** takes following values.

```
typedef enum {
   P1394_FWLL_RC_COMPLETE,
   P1394_FWLL_RC_RESERVED1,
   P1394_FWLL_RC_RESERVED2,
   P1394_FWLL_RC_RESERVED3,
   P1394_FWLL_RC_CONFLICT_ERR,
   P1394_FWLL_RC_DATA_ERR,
   P1394_FWLL_RC_TYPP1394_ERR,
   P1394_FWLL_RC_ADDRESS_ERR
} tsa1394RespCode_t;
```

P1394_FWLL_RC_COMPLETE indicates that request was successful. **pData** and **ucDataLen** are valid only if **eRespCode** is **RC_COMPLETE**.

pData and **ucDataLen** are valid only for Read block, Read Quad and lock response only. In case of Read Quad, **pData** should be typecasted to 32-bit integer to get the quadlet value. Otherwise, it contains pointer to **tsa1394MBlock_t** which contains the data.

tsa1394trDataInd_t

```
typedef struct _tsa1394trDataInd {
   tsa1394trType_t       eTrType;
   tsa1394DestOffset_t   destOffset;
   tsa1394NodeId_t       requesterId;
   tsa1394ExtTCode_t     eETrCode;
   UInt16                usDataLen;
   UInt8                 ucTrLabel;
   UInt8                 ucPriority;
   BOOL                  bBrdCast;
   tsa1394Speed_t        eCblSpeed;
   void                 *pData;
} tsa1394trDataInd_t;
```

Fields

`eTrType`	The transaction type.
`destOffset`	48-bit 1394 address.
`requesterId`	Indicates the type and ID of the requester node.
`eETrCode`	Extended transaction code valid only for lock transactions.
`usDataLen`	Indicates the byte count in **pData** for write and lock requests. In case of 'read' it indicates the number of bytes requested by the sender of this request.
`ucTrLabel`	Transaction label associated with this request. The sender expects the response with this label to match the request to the response. This should be preserved by the application and needs to be used while sending the response for this request.
`pData`	**pData** is not valid in 'read' requests. In case of write quadlet, **pData** should be typecasted to **UInt32** to get the quadlet value. In write block and lock requests, it points to **tsa1394MBlock_t** , which is described in *1394 API Data Structures* on page 124.
`eCblSpeed`	The speed at which the request is received. In some link controllers, it is not possible to find this. In which case this value is set to **P1394_FWCPHY_SPEED_SANY**.
`ucPriority`	**ucPriority** is not used and contains value 0.
`bBrdCast`	A result of a broadcast.

Description

eTrType takes following values:

```
typedef enum {
   P1394_FWTR_READBLOCK,
   P1394_FWTR_READQUAD,
   P1394_FWTR_WRITEBLOCK,
   P1394_FWTR_WRITEQUAD,
   P1394_FWTR_LOCK
} tsa1394trType_t;
```

eCblSpeed contains following values:

```
typedef {
   P1394_FWCPHY_SPEED_S100,
   P1394_FWCPHY_SPEED_S200,
   P1394_FWCPHY_SPEED_S400,
   P1394_FWCPHY_SPEED_SANY
} tsa1394Speed_t;
```

tsa1394IecTxInfo_t

```
typedef struct _tsa1394IecTxInfo {
   UInt32   trdelay;
   UInt32   maxbl;
   UInt32   pm;
   UInt32   dbs;
   UInt32   fn;
   UInt32   qpc;
   UInt32   bEnableSph;
   UInt32   fmt;
   UInt32   fdf;
   UInt32   syt;
   Char     bEnableFs;
} tsa1394IecTxInfo_t;
```

Fields

`Trdelay`	Specifies the transport delay.
`maxbl`	Specifies the maximum payload size in data blocks.
`pm`	Specifies the packing mode.
`dbs`	Specifies the data block size.
`fn`	Specifies the number of data blocks (fractions) into which each source packet is divided.
`qpc`	Specifies the quadlet packing count.
`bEnableSph`	When set, instructs the transmitter to attach a packet delivery time stamp to every application packet.
`fmt`	Specifies the format.
`fdf`	Specifies the format dependent flags.
`syt`	Specifies the format dependent flags.
`bEnableFs`	Controls enabling/disabling processing of **avf-syncin** pulses.

Description

Specifies the IEC 61883 International Standard specific packing parameters to be set for transmit mode.

tsa1394IecRxUnpackInfo_t

```
typedef struct _tsa1394IecRxUnpackInfo {
   UInt32  bpad;
   Char    bEnableFs;
} tsa1394IecRxUnpackInfo_t;
```

Fields

`bpad`	Specifies the byte padding.
`bEnableFs`	Specifies when set/cleared enables/disables processing of received syt stamps in the second CIP header quadlet and corresponding generation of **avfsyncout** pulses.

Description

Specifies the IEC 61883 International Standard specific unpacking parameters to be set for receive mode.

tsa1394IsochHdrInfo_t

```
typedef struct _tas1394IsochHdrInfo {
   UInt8   ucChannel;
   UInt8   ucTag;
   UInt8   ucSync;
   UInt8   ucSpeed;
} tsa1394IsochHdrInfo_t;
```

Fields

`ucChannel`	Specifies the channel number.
`ucTag`	Specifies the tag value. Normally the value is 0x01.
`ucSync`	Specific sync value.
`ucSpeed`	Specifies the isochronous speed.

Description

The data structure specifies the Common Isochronous Protocol header specific parameters to be set for isochronous reception.

tsa1394SbResetEventInfo_t

```
typedef struct _tsa1394SbResetEventInfo {
   long                  bwSetAside;
   tsa1394PhysNodeId_t   bmPhysId;
   tsa1394PhysNodeId_t   cmPhysId;
   Byte                  gapCount;
   tsa1394PhysNodeId_t   irmPhysId;
   tsa1394PhysNodeId_t   physId;
   tsa1394PhysNodeId_t   rootPhysId;
   Byte                  attribs;
} tsa1394SbResetEventInfo_t, *ptsa1394SbResetEventInfo_t;
```

Fields

`bwSetAside`	Bandwidth setaside.
`bmPhysId`	Bus master node's physical ID.
`cmPhysId`	Cycle master node's physical ID.
`gapCount`	Gap count, not used.
`irmPhysId`	Isochronous resource manager physical ID.
`physId`	This node's physical ID.
`rootPhysId`	Root node's physical ID.
`attribs`	Attributes. Not used.

Description

Serial bus reset event information structure.

tsa1394SbEventInfo_t

```
typedef struct _tsa1394SbEventInfo {
   union {
      tsa1394SbResetEventInfo_t   resetInfo;
   } evi;
} tsa1394SbEventInfo_t;
```

Fields

`resetInfo`	Reset Information.

Description

Serial bus event information data structure. Contains all the information, collected during self-ID read following a bus reset.

tsa1394SbmCntrlResetInitParams_t

```
typedef struct _tsa1394SbmCntrlResetInitParams {
   UInt32  ulBandWidthSetAside;
   Char    bEnableIrm;
} tsa1394SbmCntrlResetInitParams_t;
```

Fields

`ulBandWidthSetAside`	Bandwidth set aside.
`bEnableIrm`	Enable isochronous resource manager.

Description

Flags to enable/disable Isochronous Resource Manager. This structure is not applicable to the PDI1394L11 link controller.

See **tsa1394SbmCntrlReq** on page 189.

tsa1394SbmCntrlPhyConfigParams_t

```
typedef struct tsa1394SbmCntrlPhyConfigParms_t {
   tsa1394PhysNodeId_t  physNode;
   UInt16               usGapCount;
   Char                 bSetForceRoot;
   Char                 bSetGapCount;
} tsa1394SbmCntrlPhyConfigParams_t;
```

Fields

`physNode`	Physical Node ID.
`usGapCount`	Gap count.
`bSetForceRoot`	Flag which, when set, forces the node specified in physNode field to be root, after a bus reset.
`bSetGapCount`	Flag which, when set, forces the value specified in field usGapCount to be set for gapCount in the physical register.

Description

Refer to **tsa1394SbmCntrlReq** on page 189.

tsa1394SbmStatusInfo_t

```
typedef struct _tsa1394SbmStatusInfo {
   UInt32                ulBandWidthSetAside;
   tsa1394PhysNodeId_t   bmId;
   tsa1394PhysNodeId_t   cmId;
   tsa1394PhysNodeId_t   irmId;
   tsa1394PhysNodeId_t   localPhysId;
   tsa1394PhysNodeId_t   rootPhysId;
   Char                  bForceRootSet;
   UInt16                usGapCount;
} tsa1394SbmStatusInfo_t
```

Fields

`ulBandWidthSetAside`	Bandwidth set aside.
`bmId`	For BMC/IRMC only, INVALID(0x3f) otherwise.
`cmId`	For BMC/IRMC only, INVALID(0x3f) otherwise.
`irmId`	For IRMC/BMC only, INVALID(0x3f) otherwise.
`localPhysId`	Local node physical ID.
`rootPhysId`	Root node physical ID.
`bForceRootSet`	Flag indicating whether the node will be forced to be a root node on the next serial bus reset.
`usGapCount`	Gap count.

Description

Refer to **tsa1394SbmCntrlReq** on page 189.

tsa1394PhysWriteInfo_t

```
typedef struct _tsa1394PhysWriteInfo {
   tsa1394Offset_t   addrOffset;
   UInt16            dataLen;
} tsa1394PhysWriteInfo_t, *ptsa1394PhysWriteInfo_t;
```

Fields

`addrOffset`	Address offset for phyical write.
`dataLen`	Length of data.

Description

tsa1394 physical write information data structure. Not applicable to PDI1394L11 link controller.

tsa1394CtrlEventInfo_t

```
typedef struct _tsa1394CtrlEventInfo {
   union {
      tsa1394PhysWriteInfo_t   phwInfo;
      UInt32                   descrAvailable;
   } tni;
} tsa1394CtrlEventInfo_t, *ptsa1394CtrlEventInfo_t;
```

Fields

`phwInfo`	Physical write information.
`descrAvailable`	Descriptor available.

Description

tsa1394 control event information data structure. Not applicable to PDI1394L11.

tsa1394RdBlockReq_t

```
typedef struct _tsa1394RdBlockReq{
   tsa1394DestOffset_t  destOffset;
   tsa1394Speed_t       eSpeed;
   UInt8                ucPriority;
   UInt16               usDataLen;
} tsa1394RdBlockReq_t;
```

Fields

`destOffset`	Indicates the 48-bit offset on the destination node.
`usDataLen`	Length of data the sender is requesting from the remote node at the destination offset specified by **destOffset**.
`eSpeed`	Transmission speed.
`ucPriority`	Not used currently and should be set to 0.

Description

The structure above is used to send ‘Read Block’ request to the remote node.

tsa1394RdQuadReq_t

```
typedef struct _tsa1394RdQuadReq{
   tsa1394DestOffset_t   destOffset;
   tsa1394Speed_t        eSpeed;
   UInt8                 ucPriority;
} tsa1394RdQuadReq_t;
```

Fields

`destOffset`	The 48-bit 1394 address.
`eSpeed`	Transmission speed.
`ucPriority`	Priority of the request

Description

The structure is used to request quadlet data from the remote node at the destination offset specified by **destOffset**.

tsa1394WrBlockReq_t

```
typedef struct _tsa1394WrBlockReq_t {
   tsa1394DestOffset_t  destOffset;
   tsa1394Speed_t       eSpeed;
   UInt8                ucPriority;
   UInt16               usDataLen;
   tsa1394MBlock_t     *pData;
} tsa1394WrBlockReq_t;
```

Fields

destOffset	Indicates 48-bit 1394 address.
eSpeed	Transmission speed.
ucPriority	Priority.
usDataLen	Indicates the amount of data being written .
pData	The mblk pointer containing the data.

Description

This structure is used to send write block request to the destination offset specified by destOffset at the remote node.

eCblSpeed contains following values:

```
   typedef {
      P1394_FWCPHY_SPEED_S100,
      P1394_FWCPHY_SPEED_S200,
      P1394_FWCPHY_SPEED_S400,
      P1394_FWCPHY_SPEED_SANY
   } tsa1394Speed_t;
```

tsa1394WrQuadReq_t

```
typedef struct _tsa1394WrQuadReq {
   tsa1394DestOffset_t  destOffset;
   tsa1394Speed_t       eSpeed;
   UInt8                ucPriority;
   UInt32               ulData;
} tsa1394WrQuadReq_t;
```

Fields

`destOffset`	Indicates 48-bit 1394 address.
`eSpeed`	Transmission speed.
`ucPriority`	Priority.
`ulData`	Contains the quadlet data.

Description

This structure is used to send quadlet size data to the destination offset specified by **dest-Offset** at the remote node. **ulData** contains the quadlet data.

tsa1394RdBlkResp_t

```
typedef struct _tsa1394RdBlkResp {
   UInt8              ucTLabel;
   tsa1394RespCode_t  eRespCode;
   UInt16             usDataLen;
   tsa1394MBlock_t   *pData;
   tsa1394Speed_t     eSpeed;
   UInt8              ucPriority;
} tsa1394RdBlkResp_t;
```

Fields

`ucTLabel`	Transaction label associated with this request.
`eRespCode`	Contains response code which is filled by the remote node.
`usDataLen`	Indicates the data length in pData.
`pData`	Contains the data itself.
`eSpeed`	Transmission speed.
`ucPriority`	Priority.

Description

The structure is used to send Read Block response to the requester. It is expected that **ucTLabel**, **eSpeed** and **ucPriority** are copied from **tsa1394trDataInd_t** structure. **eResp-Code** should be filled to indicate the request is successful or return an error if the request cannot be completed. The values this variable takes are defined in previous sections. usDataLen and pData should be set to Null if eRespCode is any other value than **P1394_FWLL_RC_COMPLETE**. If the response code is **P1394_FWLL_RC_COMPLETE**, then pData should set to contain data to be sent and usDataLen should indicate the data length in **pData**.

tsa1394RdQuadResp_t

```
typedef struct _tsa1394RdQuadResp {
   UInt8              ucTlable;
   tsa1394RespCode_t  eRespCode;
   UInt32             ulData;
   tsa1394Speed_t     eSpeed;
   UInt8              ucPriority;
} tsa1394RdQuadResp_t;
```

Fields

`ucTlable`	Transaction label associated with this request.
`eRespCode`	Contains response code filled by the remote node.
`ulData`	Contains the quadlet value.
`eSpeed`	Transmission speed.
`ucPriority`	Priority

Description

tsa1394RdQuadResp_t structure is used to send read quadlet response to the requester. Except for **ulData**, all field are same as above. **ulData** contains the quadlet value.

tsa1394WrResp_t

```
typedef struct _tsa1394WrResp {
   UInt8              ucTLabel;
   tsa1394RespCode_t  eRespCode;
   tsa1394Speed_t     eSpeed;
   UNIT8              ucPriority;
} tsa1394WrResp_t;
```

Fields

`ucTLabel`	Transaction label associated with this request.
`eRespCode`	Contains response code filled by the remote node.
`eSpeed`	Transmission speed.
`ucPriority`	Priority.

Description

The structure is used to send write response to the requester. All fields in this structure are same as **tsa1394RdBlkResp_t**.

tsa1394LockResp_t

```
typedef struct _tsa1394LockResp {
   UInt8              ucTLabel;
   tsa1394RespCode_t  eRespCode;
   tsa1394ExtTCode_t  eETrCode;
   UInt16             usDataLen;
   tsa1394MBlock_t   *pData;
   tsa1394Speed_t     eSpeed;
   UInt8              ucPriority;
} tsa1394LockResp_t;
```

Fields

`ucTLabel`	Transaction label associated with this request.
`eRespCode`	Contains response code filled by the remote node.
`eETrCode`	Specifies the type of lock operation to be done.
`usDataLen`	Indicates the data length in pData.
`pData`	Contains the data itself.
`eSpeed`	Transmission speed.
`ucPriority`	Priority.

Description

The structure can be used to send lock response to the requester.

eETrCode has following values:

```
typedef enum {
   P1394_FWLL_ETC_MASK_SWAP,
   P1394_FWLL_ETC_COMPARE_SWAP,
   P1394_FWLL_ETC_FETCH_ADD,
   P1394_FWLL_ETC_LITTLE_ADD,
   P1394_FWLL_ETC_BOUNDED_ADD,
   P1394_FWLL_ETC_WRAP_ADD,
   P1394_FWLL_ETC_VENDOR_DEPENDENT
} tsa1394ExtTCode_t;
```

1394 API Functions

Name	Page
tsa1394GetCapabilities	158
tsaUartInstanceSetup	47
tsa1394Close	159
tsa1394GetInstanceSetup	160
tsa1394InstanceSetup	161
tsa1394AsyncCreateChannelHandle	162
tsa1394AsyncDestroyChannelHandle	162
tsa1394AsyncRegisterAddressSpace	163
tsa1394AsyncRegisterSbEvent	166
tsa1394AsyncRegisterControlEvent	168
tsa1394AsyncSendRequest	170
tsa1394AsyncSendResponse	174
tsa1394AsyncDeregisterAddressSpace	176
tsa1394AsyncDeregisterSbEvent	177
tsa1394AsyncDeregisterControlEvent	178
tsa1394IsochCreateChannelHandle	179
tsa1394IsochDestroyChannelHandle	180
tsa1394IsochSetupChannel	181
tsa1394IsochStart	182
tsa1394IsochStop	183
tsa1394SbmGetLocalNodeId	184
tsa1394SbmGetSpeed2Node	185
tsa1394SbmGetNodeIdForEuid	186
tsa1394SbmGetMaxRec	187
tsa1394SbmGetBusNodeCount	188
tsa1394SbmCntrlReq	189
tsa1394SbmGetBusId	191
tsa1394MblkInit	192
tsa1394Allocb	193

Name	Page
tsa1394AllocbFromPool	194
tsa1394EsbAlloc	195
tsa1394EsbAllocFromPool	196
tsa1394Freeb	197
tsa1394FreeMsg	198
tsa1394DupB	199
tsa1394DupMsg	200

tsa1394GetCapabilities

```
extern tmLibappErr_t tsa1394GetCapabilities (
   ptsa1394Capabilities_t   *pCap
);
```

Parameters

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

Return Codes

`P1394_FS_OK`	Success.

Description

This function returns the capabilities of the FireWire Library.

tsa1394Open

```
extern tmLibappErr_t tsa1394Open(
   Int   *instance
);
```

Parameters

`instance`	Pointer (returned) to the instance.

Return Codes

`P1394_FS_OK`	Success.
`P1394_FS_ERR_NO_MORE_INSTANCES`	No instances available.

Description

The instance returned by tsa1394Open is read-only to the application. The instance number returned to the application is a unique number and shall be used by the application in all subsequent calls to the library.

tsa1394Close

```
extern tmLibappErr_t tsa1394Close(
   Int   instance
);
```

Parameters

`instance`	A 1394 instance, as returned by **tsa1394Open**.

Return Codes

`P1394_FS_OK`	Success.
`P1394_FS_ERR_INVALID_INST`	If passed an invalid instance.

Description

This function decreases the reference count to the device each time it is called. If the reference count becomes 0 it releases all resources associated with this FireWire Device. The application-provided **cacheFreeCbFp**/**nCacheFreeCbFp** call releases any allocated memory . Refer to Section 3.5 for more details on **cacheFreeCbFp**/**nCacheFreeCbFp**. The device will have to be reopened using **tsa1394Open** call for further operations. The Firewire library maintains a reference count value for each open device.

Note
When the application which initially provided memory calls tsa1394Close, and the reference count indicates another application is using the device, the memory provided by the first application will not get freed.The application should not make assumptions about the memory it provided to the FireWire library until the application provided cacheFreeCbFp/ nCacheFreeCbFp is called.

tsa1394GetInstanceSetup

```
extern tmLibappErr_t tsa1394GetInstanceSetup(
   Int                instance,
   ptsa1394Setup_t   *setup
);
```

Parameters

`instance`	A 1394 instance, as returned by **tsa1394Open**.
`setup`	Setup info.

Return Codes

`P1394_FS_OK`	Success
`P1394_FS_ERR_INVALID_INST`	Invalid instance

Description

This function returns the default instance setup information if the device is not set up. If the device is set up it returns the current instance setup information in setupInfo.

Default Values :

setup->cacheAllocFp and **setup->nCacheAlloFp** are assigned by default to malloc.

setup->cacheFreeFp and **setup->nCacheFreeFp** are assigned by default to free.

setup->pCfgRomData is Null.

setup->ulCfgRomDataLen is 0.

setup->ulBaseAddress, **setup->ulIrq** are obtained by making a call to BSP.

setup->ulNumNodesConnected is 64.

setup->ulNumTLabels is 64.

setup->ulNumGruBufCount is 100.

The application when calling **tsa1394InstanceSetup** should set appropriate values for **pCfgRomData** and **ulCfgRomDataLen**.

tsa1394InstanceSetup

```
extern tmLibappErr_t tsa1394InstanceSetup(
   Int              instance,
   ptsa1394Setup_t  setup
);
```

Parameters

`instance`	A 1394 instance, as returned by **tsa1394Open**.
`setup`	Setup information.

Return Codes

`P1394_FS_OK`	Success
`P1394_FS_ERR_INVALID_INST`	Invalid instance
`P1394_FS_ERR_NOT_SETUP`	Instance already set up; application-specified parameters are ignored.

Description

The instance number is specified in instance. If the device is already initialized, it increases the refCount of this device. User configurable parameters are defined and passed in the structure setupInfo.

This function does hardware initialization of the FireWire device (if it is not yet initialized) in accordance to the setup information provided by the application.

tsa1394AsyncCreateChannelHandle

```
extern tmLibappErr_t tsa1394AsyncCreateChannelHandle(
   Int    instance,
   Int*   asyncChHandle
);
```

Parameters

`instance`	A 1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	An asynchronous channel handle to be created.

Return Codes

Returns a value of **tmLibappErr_t**.

Description

Create Channel Handle for the Async Operation.

tsa1394AsyncDestroyChannelHandle

```
extern tmLibappErr_t tsa1394AsyncDestroyChannelHandle(
   Int   instance,
   Int   asyncChHandle
);
```

Parameters

`instance`	A 1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Asynchronous Channel Handle.

Return Codes

Returns a value of **tmLibappErr_t**.

Description

Destroy Channel Handle for the Async Operation. This routine should be called at the end of Async operation.

tsa1394AsyncRegisterAddressSpace

```
extern tmLibappErr_t tsa1394AsyncRegisterAddressSpace(
   Int                   instance;
   Int                   asyncChHandle
   tsa1394Offset_t       startAddr,
   tsa1394Offset_t       dAddr,
   tsa1394AddrType_t     addrType,
   Bool                  isConceptual,
   tsa1394AddrReqCbFp_t  addrReqCallBk,
   void                 *userData
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Asynchronous Channel Handle.
`startAddr`	**tsa1394Offset_t** indicates 48-bit 1394 address, represented as 32-bit low portion and 16-bit high portion as ulLow and usHigh respectively.
`endAddr`	End of Address
`addrType`	Address Type, read/write/lock.
`isConceptual`	If set to T_TRUE, incoming requests whose destination offset matches with any address in registered address spaces are accepted. If set to T_FALSE, the destination offset and destination offset + data length should be within the startAddr and endAddr of the registered address space
`addrReqCallBk`	Function called with userData when incoming requests match the registered address space and transaction type.
`userData`	Application-provided data pointer.

Return Codes

`P1394_FS_OK`	Success
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel
`P1394_DSPT_ERR_INVALID_TRANS_TYPE`	Invalid transaction type passed.
`P1394_DSPT_ERR_NOCALLBACK`	Callback function pointer is Null
`P1394_DSPT_ERR_ADDRSPACE_INUSE`	Address space given was already registered.
`P1394_SYSERR_NOMEM`	Too many address spaces were registered and ran out of memory.

Description

Used to register address space with the 1394 library for a given transaction type. The call-back function is called with application data when an incoming request whose destination offset and transaction type match the registered address space.

Any address or address space can be registered only once for a transaction type. If an application tries to register address space already registered by the same or other applications, it returns an error to the caller.

startAddr and **endAddr** are 48-bit 1394 addresses (inclusive) indicating the address range. **eAddrType** indicates the type of transactions the caller is interested in getting over the specified address range. The parameter **bIsConceptual** takes two values: TRUE and FALSE. If set to TRUE, incoming requests whose destination offset matches any address in registered address spaces are accepted. If set to FALSE, the destination offset and destination offset+data length should be within the **startAddr** and **endAddr** of the registered address space. Otherwise, the 1394 library rejects the incoming request by sending an error response to the remote node. The structure **tsa1394Offset_t** has the following format:

```
typedef struct {
   UInt32 ulLow;
   UInt16 usHigh;
} tsa1394Offset_t;
```

tsa1394Offset_t indicates a 48-bit 1394 address, represented as 32-bit low portion and 16-bit high portion as ulLow and usHigh respectively.

The enum **tsa1394AddrType_t** has following values defined:

```
typedef struct {
   P1394_FW_ADDR_READ,
   P1394_FW_ADDR_WRITE,
   P1394_FW_ADDR_LOCK
} tsa1394AddrType_t;
```

This is its callback format:

```
void AppTransIndCallback (
   tsa1394AppContext_t   asyncChannel,
   tsa1394trDataInd_t   *dataInd,
   void                 *pUserData
)
```

Description

The 1394 library calls the application-supplied callback function with **asyncChannel** indicating the asynchronous channel, **dataInd** which has transaction parameters corresponding to the incoming request, and **pUserData** which was passed while registering the address space.

Note
The application should call tsa1394FreeMsg on dataInd->pData, on an incoming read response, after it has processed the data. Refer to section 7.8 on more information about tsa1394FreeMsg.

The structure **tsa1394trDataInd_t** has following parameters:

```
typedef struct {
   tsa1394trType_t       eTrType;
   tsa1394DestOffset_t   destOffset;
   tsa1394NodeId_t       requesterId;
   tsa1394ExtTCode_t     eETrCode;
   UInt16                usDataLen;
   UInt8                 ucTrLabel;
   UInt8                 ucPriority;
   BOOL                  bBrdCast;
   tsa1394Speed_t        eCblSpeed;
   void                 *pData;
} tsa1394trDataInd_t;
```

eTrType indicates the transaction type and takes following values:

```
typedef enum {
   P1394_FWTR_READBLOCK,
   P1394_FWTR_READQUAD,
   P1394_FWTR_WRITEBLOCK,
   P1394_FWTR_WRITEQUAD,
   P1394_FWTR_LOCK
} tsa1394trType_t;
```

destOffset indicates a 48-bit 1394 address. **requesterId** is the 16-bit node ID that initiated the request. **eETrCode** is an extended transaction code and is valid only for lock transactions. **ucTrLable** is transaction label associated with this request. The sender expects the response with this label to match the request to the response. This should be preserved by the application and needs to be used while sending a response for this request. **usDataLen** indicates the byte count in pData for write and lock requests. In case of 'read' it indicates the number of bytes requested by the sender of this request. pData is not valid in a 'read' request. **ucPriority** is not used and contains value 0. **eCblSpeed** indicates the speed at which the request is received. The boolean field **bBrdCast** indicates if the indication has arrived as a result of a broadcast. In some link controllers it is not possible to find out at which speed the request is received. In this case, this value is set to **P1394_FWCPHY_SPEED_SANY**.

eCblSpeed contains following values

```
typedef {
   P1394_FWCPHY_SPEED_S100,
   P1394_FWCPHY_SPEED_S200,
   P1394_FWCPHY_SPEED_S400,
   P1394_FWCPHY_SPEED_SANY
} tsa1394Speed_t;
```

In case of write quadlet, **pData** should be typecasted to **UInt32** to get the quadlet value. In write block and lock requests, it points to **tsa1394MBlock_t**, which is described in *1394 API Data Structures* on page 124.

tsa1394AsyncRegisterSbEvent

```
extern tmLibappErr_t tsa1394AsyncRegisterSbEvent(
   Int                    instance,
   Int                    asyncChHandle,
   tsa1394SbEvent_t       eventId,
   tsa1394SbEventCbFp_t   sbEventCallBk,
   void                  *userData
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Asynchronous Channel Handle.
`eventId`	Event ID.
`sbEventCallBk`	Function called by the library when the event occurs.
`userData`	Application Data.

Return Codes

`P1394_FS_OK`	Success
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel
`P1394_DSPT_ERR_INVALID_SBEVENT`	Invalid **sbEvent** value is passed.
`P1394_DSPT_ERR_NOCALLBACK`	Callback function passed is Null.
`P1394_DSPT_ERR_DUPLICATE_SBEVENT`	This **sbevent** is already registered in this context.
`P1394_SYSERR_NOMEM`	Too many registrations, out of memory.

Description

This function is used to register callback function for serial bus events. **asyncChannel** indicates the asynchronous channel which is interested in serial bus events. **instance** indicates the FireWire device which the application is interested in. **sbEvent** indicates the interested serial bus event. **pSbEvCallback** is an application function, which will be called when serial bus event occurs. **pUserData** will be passed to callback function.

Only one registration for **sbEvent** is allowed in one asynchronous channel.

Using this function, the application can register its functions for following bus events.

```
P1394_FWSBM_EV_OCCUPANCY_VIOLATION
P1394_FWSBM_EV_RESET_START
P1394_FWSBM_EV_RESET_COMPLETE
P1394_FWSBM_EV_CYCLE_TOO_LONG
P1394_FWSBM_EV_POWER_FAIL
P1394_FWSBM_EV_DUPLICATE_CHANNEL
P1394_FWSBM_EV_CRC_ERROR
P1394_FWSBM_EV_REQ_DATA_ERROR
P1394_FWSBM_EV_RESP_ACK_MISSING
P1394_FWSBM_EV_RESP_DATA_ERROR
P1394_FWSBM_EV_RESP_FORMAT_ERROR
P1394_FWSBM_EV_RESP_RETRY_FAILED
P1394_FWSBM_EV_UNEXPECTED_CHANNEL
P1394_FWSBM_EV_UNKNOWN_TRANS_CODE
P1394_FWSBM_EV_UNSOLICITED_RESPONSE
```

Most of the applications require only RESET_START and RESET_COMPLETE events to be notified to them. Reset of bus events are normally used by management applications.

This is its callback format:

```
void AppSbEventIndCallback (
   tsa1394AppContext_t  asyncChannel,
   tsa1394SbEvent_t     sbEvent,
   void                *pUserData
)
```

This function is called by the 1394 library to report bus events. **asyncChannel** indicates the context of the application which registered this callback function. **sbEvent** is passed to the callback function indicating the bus event (as specified above) occurred. **event-Data** is not used.

tsa1394AsyncRegisterControlEvent

```
extern tmLibappErr_t tsa1394AsyncRegisterControlEvent(
   Int                     instance,
   Int                     asyncChHandle,
   tsa1394CtrlEvent_t      eventId,
   tsa1394CtrlEventCbFp_t  ctrlEventCallBk,
   void                   *userData
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Handle for the asynchronous channel acquired from **tsa1394AsyncCreateChannelHandle**.
`eventId`	Event ID.
`ctrlEventCallBk`	Function called by the library when event occurs.
`userData`	Application's data pointer to be passed into the **ctrlEventCallBk**.

Return Codes

`P1394_FS_OK`	Success
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel.
`P1394_DSPT_ERR_INVALID_SBEVENT`	Invalid control event specified.
`P1394_DSPT_ERR_NOCALLBACK`	Callback function pointer is Null.
`P1394_SYSERR_NOMEM`	Out of memory.

Description

Used to register control events such as notification of physical write and resource availability in HAL (Hardware layer). **asyncChannel** indicates the application which is interested in control event specified by **controlEvent**. **instance** indicates the FireWire instance number which the application is interested in. **pCtrlCbFp** is the function to be called when **controlEvent** is generated in 1394 library. **pUserData** is the application-specified data and will be passed to callback function.

The following control events are supported in this call.

```
   typedef enum {
      P1394_FWCE_PHYSICAL_WRITE,
      1394_FWCE_RESOURCE_AVAILABILITY
   } tsa1394CtrlEvent_t;
```

The **P1394_FWCE_PHYSICAL_WRITE** event can be generated only if Link controller supports physical write transactions and generated interrupt on physical write.

The **P1394_FWCE_RESOURCE_AVAILABILITY** event is generated when the HAL layer gets resources to send the transaction packets out. This event is required because there could be more than one application, or an application which is generating lot of traffic on 1394 library. The number of resources allocated to HAL to send packets out are limited. When these resources are used, any call to 1394 library to send the packets will fail. At this time, applications can register for this event with the 1394 library. When the resources are available in HAL, the 1394 library calls the application-supplied callback function and the application can start sending packets again. In this case, after calling the application-supplied function, the 1394 library removes the registration. Applications are required to register with this event only if they cannot send packets out due to HAL resource crunch.

This is its callback format:

```
void AppControlEventCallback (
   tsa1394AppContext_t  asyncChannel,
   sa1394CtrlEvent_t    controlEvent,
   void                *pUserData
)
```

This function is called by the 1394 library when the control event occurs. **asyncChannel** indicates the asynchronous channel which registered this callback function for control event specified as **controlEvent**. **pUserData** is a callback function argument which is passed using registration call.

tsa1394AsyncSendRequest

```
extern tmLibappErr_t tsa1394AsyncSendRequest(
   Int                  instance,
   Int                  asyncChHandle,
   tsa1394NodeId_t      nodeId,
   tsa1394trType_t      transType,
   void                *transData,
   tsa1394AtReqCbFp_t   atReqCallBk,
   void                *userData
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Application sending the TransType asynchronous request to the node specified by nodeId.
`nodeId`	Node ID.
`transType`	Read/write block, read/write quad etc.
`transData`	Information associated with the request.
`atReqCallBk`	Function called by the library when request completes.
`userData`	Application-supplied data passed by the 1394 library to the callback function.

Return Codes

`P1394_FS_OK`	Success
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel.
`P1394_GENERR_NULLARG`	Null argument is passed.
`P1394_DSPT_INVALID_EXTENDED_CODE`	Invalid extended code.
`P1394_SYSERR_NOMEM`	No memory.
`P1394_TR_ERR_IN_RESET_MODE`	Bus reset processing.
`P1394_TR_ERR_UNKNOWN_NODE`	Unknown destination physical ID.
`P1394_TR_ERR_UNKNOWN_BUS`	Unknown bus.
`P1394_TR_ERR_DATALEN_MISMATCH`	Payload in the request packet exceeds the maximum.
`P1394_TR_ERR_SPEED_MISMATCH`	Speed mismatch.
`P1394_TR_ERR_OUTOF_TLABELS`	Out of transactions labels.
`HAL_INVALID_DATALEN`	Invalid data length.

Description

This function is called to send an asynchronous request to the remote node. **asyncChannel** indicates the application sending the asynchronous request of type **eTransType** to the node specified by nodeId. **instance** indicates the FireWire instance number which the application is calling. **pTransData** contains the information associated with the request. **pAtRqCbFp** specifies the function pointer which is called with the response received from the remote node to this request. **pUserData** is application supplied data which will be passed by the 1394 library to the callback function.

tsa1394NodeId_t is defined as **unsigned short** and should have 'busid' and 'phsical id.'

The enum **tsa1394trType_t** has following values.

```
typedef enum {
   P1394_FWTR_READBLOCK,
   1394_FWTR_READQUAD,
   P1394_FWTR_WRITEBLOCK,
   P1394_FWTR_WRITEQUAD,
   P1394_FWTR_LOCK
} tsa1394trType_t
```

P1394_FWTR_READBLOCK is used to request block of data from the remote node.

P1394_FWTR_READ_QUAD is used to request 4 bytes of data from the remote node.

P1394_FWTR_WRITEBLOCK is used to send block of data to the remote node, whereas **P1394_FWTR_WRITEQUAD** can be used to send 4 bytes of data to the remote node.

P1394_FWTR_LOCK is used to perform lock operation on the remote node.

pTransData contains different information based on the transaction type. The following structures describe information for each transaction type.

```
typedef struct {
   tsa1394DestOffset_t  destOffset;
   tsa1394Speed_t       eSpeed;
   UInt8                ucPriority;
   UInt16               usDataLen;
} tsa1394RdBlockReq_t;
```

The structure above is used to send 'Read Block' request to the remote node. **destOffset** indicates the 48-bit offset on the destination node. **usDataLen** indicates the length of data, the sender is requesting from the remote node at the destination offset specified by **destOffset**. **eSpeed** indicates the speed, the packet should be sent. **ucPriority** is not used currently and should be set to 0.

```
typedef struct {
   tsa1394DestOffset_t  destOffset;
   tsa1394Speed_t       eSpeed;
   UInt8                ucPriority;
} tsa1394RdQuadReq_t;
```

The structure above is used to request quadlet data from the remote node at the destination offset specified by **DestOffset**.

```
typedef struct {
   tsa1394DestOffset_t  destOffset;
   tsa1394Speed_t       eSpeed;
   UInt8                ucPriority;
   UInt16               usDataLen;
```

```
      tsa1394MBlock_t      *pData;
   } tsa1394WrBlockReq_t;
```

The structure above is used to send write block request to the destination offset specified by **destOffset** at the remote node. **usDataLen** indicates the amount of data being written and **pData** is mblk pointer containing the data.

```
   typedef struct {
      tsa1394DestOffset_t  destOffset;
      tsa1394Speed_t       eSpeed;
      UInt8                ucPriority;
      UInt32               ulData;
   } tsa1394WrQuadReq_t;
```

The structure above is used to send quadlet size data to the destination offset specified by **destOffset** at the remote node. **ulData** contains the quadlet data.

```
   typedef struct {
      tsa1394DestOffset_t  destOffset;
      tsa1394Speed_t       eSpeed;
      UInt8UInt16          ucPriority;
      tsa1394ExtTCode_t    eETrCode;
      UInt16               usDataLen;
      tsa1394MBlock_t     *pData;
   } tsa1394LockReq_t;
```

The structure above is used to send lock request to the destination offset specified by **destOffset** at the remote node. **usDataLen** contains the data to be sent in lock request. **pData** should contain the data itself. **eETrCode** specifies the type of lock operation to be done and has following values.

```
   typedef enum {
      P1394_FWLL_ETC_MASK_SWAP,
      P1394_FWLL_ETC_COMPARE_SWAP,
      P1394_FWLL_ETC_FETCH_ADD,
      P1394_FWLL_ETC_LITTLE_ADD,
      P1394_FWLL_ETC_BOUNDED_ADD,
      P1394_FWLL_ETC_WRAP_ADD,
      P1394_FWLL_ETC_VENDOR_DEPENDENT
   } tsa1394ExtTCode_t;
```

This is its callback format:

```
   void AppRecvRespCallback (
      tsa1394AppContext_t   asyncChannel,
      tsa1394trDataCnfm_t  *pDataCnf,
      void              *pUserData
   )
```

This function is called by the 1394 library once it receives the response for the request sent. **asyncChannel** indicates the asynchronous channel which sent the request. **dataCnf** contains information related to the response received.

The structure **tsa1394trDataCnfm_t** has following fields.

```
   typedef struct {
      tsa1394trReqStatus_t   eTrStatus;
      tsa1394RespCode_t      eRespCode;
      tsa1394MBlock_t       *pData
      UInt16                 usDataLen;
   tsa1394trDataCnfm_t;
```

eTrStatus contains the transaction status. It takes following values.

```
typedef enum {
   P1394_FWTL_RSTAT_COMPLETE,
   P1394_FWTL_RSTAT_TIMEOUT,
   P1394_FWTL_RSTAT_ACK_MISS,
   P1394_FWTL_RSTAT_DATA_ERR
} tsa1394trReqStatus_t;
```

Other fields in **tsa1394trDataCnfm_t** are valid only if **eTrStatus** is **P1394_FWTL_RSTAT_COMPLETE**. **P1394_FWTL_RSTAT_TIMEOUT** indicates there is no response to the request sent. **P1394_FWTL_RSTAT_ACK_MISS** and **P1394_FWTL_RSTAT_DATA_ERR** indicate a problem was encountered while sending request to the remote node.

eRespCode contains response code which is filled by the remote node. It takes following values.

```
typedef enum {
   P1394_FWLL_RC_COMPLETE,
   P1394_FWLL_RC_RESERVED1,
   P1394_FWLL_RC_RESERVED2,
   P1394_FWLL_RC_RESERVED3,
   P1394_FWLL_RC_CONFLICT_ERR,
   P1394_FWLL_RC_DATA_ERR,
   P1394_FWLL_RC_TYPE_ERR,
   P1394_FWLL_RC_ADDRESS_ERR
} tsa1394RespCode_t;
```

P1394_FWLL_RC_COMPLETE indicates that request was successful. **pData** and **ucDataLen** are valid only if **eRespCode** is **RC_COMPLETE**.

pData and **ucDataLen** are valid only for Read block, Read Quad, and lock response. In case of Read Quad, **pData** should be typecasted to 32-bit integer to get the quadlet value. Otherwise, it contains pointer to 'mblk' which contains the data.

tsa1394AsyncSendResponse

```
extern tmLibappErr_t tsa1394AsyncSendResponse(
   Int                instance,
   Int                asyncChHandle,
   tsa1394NodeId_t    nodeId,
   tsa1394trType_t    transType,
   void              *response
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Application which is sending the response.
`nodeId`	Node idenfification number.
`transType`	Transaction type.
`response`	Information an application sends as part of response.

Return Codes

`P1394_FS_OK`	Success
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel.
`P1394_DSPT_ERR_INVALID_TRANS_TYPE`	Invalid transaction type is passed.
`P1394_DSPT_ERR_ADDRESS_NOT_REGISTERED`	Address specified was not registered.

Description

This function sends a response to the request received from the remote node. **asyncChannel** indicates the application which is sending the response. It should be same as **asyncChannel** received as part of incoming request. **instance** indicates the FireWire instance number which the application is interested in. **nodeId** is the node ID of the requester and **pResponse** should contains information the application wants to send as part of a response. The response information depends on transaction type **eTransType**. **eTransType** should be the same as **eTrType** of **tsa1394trDataInd_t**. **nodeId** should be the same as **requesterId** of **tsa1394trDataInd_t**.

pResponse takes following structure based on eTrType.

```
   typedef struct {
      UInt8              ucTLabel;
      tsa1394RespCode_t  eRespCode;
      UInt16             usDataLen;
      tsa1394MBlock_t   *pData;
      tsa1394Speed_t     eSpeed;
      UInt8             ucPriority;
   } tsa1394RdBlkResp_t;
```

tsa1394RdBlkResp_t structure is used to send Read Block response to the requester. It is expected that **ucTLabel**, **eSpeed** and **ucPriority** are copied from **tsa1394trDataInd_t** structure. **eRespCode** should be filled to indicate the request is successful, or return an error if request cannot be completed. The values this variable takes are defined in previous sections. **usDataLen** and **pData** should be set to Null if **eRespCode** is any other value than **P1394_FWLL_RC_COMPLETE**. If the response code is **P1394_FWLL_RC_COMPLETE**, then **pData** should set to contain data to be sent and **usDataLen** should indicate the data length in **pData**.

```
   typedef struct {
      UInt8             ucTlable;
      tsa1394RespCode_t  eRespCode;
      UInt32            ulData;
      tsa1394Speed_t     eSpeed;
      UInt8             ucPriority;
   }tsa1394RdQuadResp_t;
```

tsa1394RdQuadResp_t structure is used to send read quadlet responses to the requester. Except for **ulData**, all fields are the same as above. **ulData** contains the quadlet value.

```
   typedef struct {
      UInt8   ucTLabel;
      tsa1394RespCode_t    eRespCode;
      tsa1394Speed_t   eSpeed;
      UNIT8   ucPriority;
   } tsa1394WrResp_t;
```

tsa1394WrResp_t is used to send write responses to the requester. All fields in this structure are same as **tsa1394RdBlkResp_t**.

```
   typedef struct {
      UInt8             ucTLabel;
      tsa1394RespCode_t  eRespCode;
      tsa1394ExtTCode_t  eETrCode;
      UInt16            usDataLen;
      tsa1394MBlock_t   *pData;
      tsa1394Speed_t     eSpeed;
      UInt8             ucPriority;
   } tsa1394LockResp_t;
```

tsa1394LockResp_t can be used to a send lock responses to the requester.

tsa1394AsyncDeregisterAddressSpace

```
extern tmLibappErr_t tsa1394AsyncDeregisterAddressSpace (
   Int                instance,
   Int                asyncChHandle,
   tsa1394Offset_t    address,
   tsa1394AddrType_t  addrType
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Async Channel Handle.
`address`	Start Address of the address space.
`addrType`	Address Type: READ, WRITE, or LOCK.

Return Codes

`P1394_FS_OK`	Success
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel.
`P1394_DSPT_ERR_INVALID_TRANS_TYPE`	Invalid transaction type is passed.
`P1394_DSPT_ERR_ADDR_NOT_REGISTERED`	Address specified was not registered.

Description

Applications can use this function to **deregister** address space which was registered before using **tsa1394AsyncRegisterAddressSpace**. Address space is uniquely identified by **startAddr** and type of transactions supported as indicated by the address type **eAddrType**. instance indicates the FireWire instance number which the application is interested in.

The structure **tsa1394Offset_t** has the format.

```
typedef struct {
   UInt32   ulLow;
   UInt16   usHigh;
} tsa1394DestOffset_t;
typedef tsa1394DestOffset_t tsa1394Offset_t;
```

tsa1394Offset_t indicates a 48-bit 1394 address, represented as 32-bit low portion and 16-bit high portion as ulLow and usHigh respectively.

The enum **tsa1394AddrType_t** has following values defined:

```
typedef enum {
   P1394_FW_ADDR_READ,
   P1394_FW_ADDR_WRITE,
   P1394_FW_ADDR_LOCK
}tsa1394AddrType_t;
```

tsa1394AsyncDeregisterSbEvent

```
extern tmLibappErr_t tsa1394AsyncDeregisterSbEvent(
   Int                instance,
   Int                asyncChHandle,
   tsa1394SbEvent_t   event
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Asynchronous Channel Handle.
`event`	Serial Bus Event, previously registered.

Return Codes

`P1394_FS_OK`	Success.
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel.
`P1394_DSPT_ERR_INVALID_SBEVENT`	Invalid serial bus event.
`P1394_DSPT_ERR_NOTREGISTERED`	Serial bus event was not registered.

Description

This function is used to deregister previously registered serial bus event **sbEvent** for the asynchronous channel **asyncChannel**. **instance** indicates the FireWire instance number which the application is querying.

tsa1394AsyncDeregisterControlEvent

```
extern tmLibappErr_t tsa1394AsyncDeregisterControlEvent(
   Int                 instance,
   Int                 asyncChHandle,
   tsa1394CtrlEvent_t  eventId
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`asyncChHandle`	Asynchronous channel handle.
`eventId`	Event ID to deregister.

Return Codes

`P1394_FS_OK`	Success.
`P1394_DSPT_INVALID_APPCONTEXT`	Invalid asynchronous channel.
`P1394_DSPT_ERR_INVALID_SBEVENT`	Invalid control event.
`P1394_DSPT_ERR_NOTREGISTERED`	Control event was not registered.

Description

This function is used to deregister a previously registered control event **controlEvent** for asynchronous channel **asyncChannel**.

The following control events are supported in this call.

```
typedef enum {
   P1394_FWCE_PHYSICAL_WRITE,
   P1394_FWCE_RESOURCE_AVAILABILITY
} tsa1394CtrlEvent_t;
```

tsa1394IsochCreateChannelHandle

```
extern tmLibappErr_t tsa1394IsochCreateChannelHandle(
   Int    instance,
   Int   *isochChHandle
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`isochChHandle`	Isochronous channel handle.

Return Codes

Returns a value of **tmLibappErr_t**.

Description

Creates an Isochronous channel handle.

tsa1394IsochDestroyChannelHandle

```
extern tmLibappErr_t tsa1394IsochDestroyChHandle(
   Int   instance,
   Int   isochChHandle
);
```

Parameters

`instance`	A 1394 instance, as returned by **tsa1394Open**.
`isochChHandle`	Isochronous Channel Handle.

Return Codes

Returns a value of **tmLibappErr_t**.

Description

Destroys an isochronous channel handle.

tsa1394IsochSetupChannel

```
extern tmLibappErr_t tsa1394IsochSetupChannel(
   Int                     instance,
   Int                     isochChHandle,
   UInt8                   txRxFlag,
   tsa1394IsochHdrInfo_t*   isochHeader,
   void                   *pIecPackInfo
);
```

Parameters

`instance`	A 1394 instance, as returned by **tsa1394Open**.
`isochChHandle`	Isochronous Channel Handle.
`txRxFlag`	Either **P1394_ISOCH_TX_MODE** or **P1394_ISOCH_RX_MODE.**
`isochHeader`	pointer to **tsa1394IsochHdrInfo_t.**
`pIecPackInfo`	**tsa1394IecTxInfo_t**, or **tsa1394IecRxUnpackInfo_t.**

Return Codes

Returns a value of **tmLibappErr_t**.

Description

Sets up an Isochronous Channel.

tsa1394IsochStart

```
extern tmLibappErr_t tsa1394IsochStart(
   Int   instance,
   Int   isochChHandle
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`isochChHandle`	Isochronous Channel Handle.

Return Codes

`P1394_FS_OK`	Success.

Description

The application calls **tsa1394IsochStart** for starting the transmission or reception.

tsa1394IsochStop

```
extern tmLibappErr_t tsa1394IsochStop(
   Int   instance,
   Int   isochChHandle
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`isochChHandle`	Isochronous Channel Handle.

Return Codes

`P1394_FS_OK`	Success.

Description

Stop the Isochronous operation.

tsa1394SbmGetLocalNodeId

```
extern tmLibappErr_t tsa1394SbmGetLocalNodeId(
   Int                 instance,
   tsa1394NodeId_t    *pNodeId
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`pNodeId`	Node idenfification number.

Return Codes

`P1394_FS_OK`	Success.
`P1394_SBM_ERR_IN_RESET`	The 1394 bus is in Reset mode

Description

Gets the value of the local node ID. This is returned in a **tsa1394Node**. **instance** indicates the FireWire instance number which the application is querying.

tsa1394SbmGetSpeed2Node

```
extern tmLibappErr_t tsa1394SbmGetSpeed2Node(
   Int               instance,
   tsa1394NodeId_t   nodeId,
   tsa1394Speed_t   *pSpeed
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`nodeId`	Node idenfification number.
`pSpeed`	Transmission speed at which the packet is sent.

Return Codes

`P1394_FS_OK`	Success.
`P1394_SBM_INVALID_FWDRV`	Invalid device.
`P1394_SBM_ERR_IN_RESET`	Reset error.
`P1394_SBM_SPEED_UNKNOWN`	Speed unknown.

Description

Returns the speed between the current node, whose instance is instance and **atsa1394Node** in **pSpeed**.

tsa1394SbmGetNodeIdForEuid

```
extern tmLibappErr_t tsa1394SbmGetNodeIdForEuid(
   Int                instance,
   tsa1394EUId_t      EuId,
   tsa1394NodeId_t   *pNodeId
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`EuId`	Extended Unique Identifier, 64 bits, as defined by the IEEE.
`pNodeId`	Node idenfification number.

Return Codes

`P1394_FS_OK`	Success.
`P1394_SBM_INVALID_FWDRV`	Invalid device.
`P1394_SBM_ERR_IN_RESET`	Reset error.
`P1394_SBM_EUID_UNKNOWN`	Unknown EUID.

Description

Returns the node ID of instance in aNode.

tsa1394SbmGetMaxRec

```
extern tmLibappErr_t tsa1394SbmGetMaxRec(
   Int                instance,
   tsa1394NodeId_t    nodeId,
   UInt16            *pMaxRec
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`nodeId`	Node idenfification number.
`pMaxRec`	an input/output field to hold the max record value read from config rom data.

Return Codes

`P1394_FS_OK`	Success.
`P1394_SBM_INVALID_FWDRV`	Invalid device.
`P1394_SBM_ERR_IN_RESET`	Reset error.
`P1394_SBM_MAXREC_UNKNOWN`	Maxrec unknown.

Description

The maximum record length of the specified node, whose instance is instance is returned in **aUsMaxRecLen**.

tsa1394SbmGetBusNodeCount

```
extern tmLibappErr_t tsa1394SbmGetBusNodeCount(
   Int       instance,
   UInt32   *pNodeCount
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`pNodeCount`	pointer to Node Count.

Return Codes

`P1394_FS_OK`	Success.
`P1394_SBM_INVALID_FWDRV`	Invalid device.
`P1394_SBM_ERR_IN_RESET`	Reset error.

Description

The number of nodes on the bus will be returned in **aNodeCount**.

tsa1394SbmCntrlReq

```
extern tmLibappErr_t tsa1394SbmCntrlReq(
   UInt32               instance,
   tsa1394SbmCntrlOp_t  controlCommand,
   void                *controlParams
);
```

Parameters

`instance`	The 1394 instance, as returned by **tsa1394Open**.
`controlCommand`	Control command: one of the following: `E_SBM_CNTRL_RESET,` `E_SBM_CNTRL_INIT,` `E_SBM_CNTRL_LINKON,` `E_SBM_CNTRL_STATUS,` `E_SBM_CNTRL_PHYCONFIG.`
controlParams	Input parameter, if any.

Return Codes

`P1394_FS_OK`	Success.
`P1394_SBM_INVALID_FWDRV`	Invalid device.
`P1394_SBM_ERR_IN_RESET`	Reset error.
`P1394_SBM_INVALID_CNTRLOP`	Invalid control loop.

Description

tsa1394SbmCntrlOp_t defines the possible control requests that can be requested:

```
typedef enum {
   P1394_SBM_CNTRL_RESET,
   P1394_SBM_CNTRL_INIT,
   P1394_SBM_CNTRL_LINKON,
   P1394_SBM_CNTRL_STATUS,
   P1394_SBM_CNTRL_PHYCONFIG
} tsa1394SbmCntrlOp_t;
```

Depending on the type of Control operation selected, appropriate parameters need to be passed. If **P1394_SBM_CNTRL_RESET** or **P1394_SBM_CNTRL_INIT** is selected, **aCntrlParams** will point to the structure defined below. **ulBandWidthSetAside** and **bEnableIrm** should be 0.

```
typedef struct SbmCntrlResetInitParams_t {
   UInt32  ulBandWidthSetAside;
   BOOL    bEnableIrm;
} SbmCntrlResetInitParams_t;
```

If **P1394_SBM_CNTRL_LINKON** is selected, **aCntrlParams** will be the actual PhysId value of the node whose Link needs to be activated.

If **P1394_SBM_CNTRL_STATUS** is selected, **aCntrlParams** will point to the structure defined below, and the values will be returned at the appropriate locations in the structure.

```
typedef struct _SbmStatusInfo_t {
   UInt32               ulBandWidthSetAside;
   tsa1394PhysNodeId_t  bmId;
/* For BMC / IRMC only, INVALID(Øx3f) otherwise */
   tsa1394PhysNodeId_t  cmId;
/* For BMC / IRMC only, INVALID(Øx3f) otherwise */
   tsa1394PhysNodeId_t  irmId;
/* For IRMC / BMC only, INVALID(Øx3f) otherwise */
   tsa1394PhysNodeId_t  localPhysId;
   tsa1394PhysNodeId_t  rootPhysId;
   BOOL                 bForceRootSet;
   UInt16               usGapCount;
} SbmStatusInfo_t
```

If **P1394_SBM_CNTRL_PHYCONFIG** is selected, **aCntrlParams** will point to the structure defined below, and the appropriate values as required need to be programmed.

```
typedef struct SbmCntrlPhyConfigParms_t {
   tsa1394PhysNodeId_t   physNode;
   UInt16                usGapCount;
   BOOL                  bSetForceRoot;
   BOOL                  bSetGapCount;
}SbmCntrlPhyConfigParams_t;
```

Depending on the parameter passed, appropriate actions are taken.

tsa1394SbmGetBusId

```
extern UInt32 tsa1394SbmGetBusId(
   UInt32   instance,
   UInt16  *pBusId
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`pBusId`	Input/output parameter to hold the Bus ID.

Return Codes

`P1394_FS_OK`	Success.
`P1394_SBM_INVALID_FWDRV`	Invalid device.

Description

The bus ID is returned in **aBusId**.

tsa1394MblkInit

```
extern tmLibappErr_t tsa1394MblkInit(
   Int                        instance,
   long                      *lPoolId,
   Char                      *pPoolName,
   Byte                      *pBuffer,
   long                       lBufLength,
   long                       lNoMblks,
   tsa1394MblkBufConfig_t    *pBufCfg,
   short                      iAlignment,
   Bool                       bDefault,
   void                     (*pFreeFn)(void*),
   void                      *pFreeFnArg
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`lPoolId`	Returned ID of the memory pool to be processed.
`pPoolName`	Name of the memory pool.
`pBuffer`	The application-provided buffer to be configured.
`lBufLength`	The length of the application-provided buffer.
`lNoMblks`	Total number of all **ulBuffers** in the array of **tsa1394MblkBufConfig_t**.
`pBufCfg`	Desired configuration of the memory pool.
`iAlignment`	Alignment of the memory blocks, e.g., 64 indicating 64 byte aligned blocks.
`bDefault`	Boolean flag indicating whether this pool is a default pool.
`pFreeFn`	Function to free the entire memory.
`pFreeFnArg`	The pointer to the argument to be passed to **pFreeFn**.

Return Codes

`P1394_FS_OK`	Success

Description

This function provides a mechanism to organize a data buffer into a set of memory blocks in **tsa1394MBlock_t**.

The function **pFreeFn** if set to a valid value by the application, will be called with **pFreeFnArg**, when a mblock is returned (freed) to the mblock pool using **tsa1394FreeMsg** or **tsa1394Freeb**. This provides a mechanism to inform the application that mblocks are available for application usage.

tsa1394Allocb

```
extern tmLibappErr_t tsa1394Allocb(
   Int               instance,
   long              lSize,
   tsa1394MBlock_t  **retBlk
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`lSize`	The size of the requested buffer from the default pool.
`retBlk`	Pointer a pointer to the allocated block.

Return Codes

`P1394_FS_OK`	Success.

Description

A pointer to the allocated **tsa1394MBlock_t** is returned. The size of the buffer to be allocated is **lSize,** which is an input parameter. **lSize** is the size of the requested buffer from the default pool. The buffer allocated is equal to or greater than the size requested. If the default pool is not set up, this returns an error.

tsa1394AllocbFromPool

```
extern tmLibappErr_t tsa1394AllocbFromPool(
   Int                 instance,
   long                lPoolId,
   long                lSize,
   tsa1394MBlock_t   **retBlk
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`lPoolId`	ID of the memory pool where the mblock is to be allocated.
`lSize`	The size of the requested buffer.
`*retBlk`	Pointer to a pointer to the returned memory block.

Return Codes

`P1394_FS_OK`	Success.

Description

The function allocates a memory block from a memory pool.

tsa1394EsbAlloc

```
extern tmLibappErr_t tsa1394EsbAlloc(
   Int                 instance,
   Byte               *pBase,
   long                lSize,
   tsa1394FreeRtn_t   *pFrtn,
   tsa1394MBlock_t   **retBlk
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`pBase`	Pointer to the base of the data passed by the application.
`lSize`	The size of the data.
`pFrtn`	Pointer to the routine called for freeing the data block.
`*retBlk`	Pointer to the returned memory block.

Return Codes

`P1394_FS_OK`	Success.

Description

In **tsa1394EsbAlloc**, the data buffer is given by the application. If no pool is set as the default pool, this function will return error.

An mblock structure is returned:

lPoolId has the pool ID information.

pNext is Null.
pCont is Null.
pRptr = pBase.
pWptr = pBase.
pData->lPoolId stands for the pool ID.
pData->pDbFreep is used internally, set to Null when application obtains an mblock.
pData->pDbBase = pBase.
pData->pDbLim = pBase + lSize.
pData->ulDbRef = 0.
pData->ucDbType is used internally, this field should not be modified by the application.
pData ->DbFrtn = pFrtn.
ulRequestedCount = lSize.

Difference between tsa1394EsbAlloc and tsa1394Allocb

tsa1394EsbAlloc is typically called by an application when it provides its own data buffer and requires only an mblock header from the mblock pool.

pFrtn will be called when the data block is freed.

tsa1394EsbAllocFromPool

```
extern tmLibappErr_t tsa1394EsbAllocFromPool(
   Int                  instance,
   long                 lPoolId,
   Byte                *pBase,
   long                 lSize,
   tsa1394FreeRtn_t    *pFrtn,
   tsa1394MBlock_t    **retBlk
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`lPoolId`	Identification of the pool used for allocation
`pBase`	Pointer to the base of the data passed by the application.
`lSize`	The size of data.
`pFrtn`	Pointer to the routine to be called for freeing the data buffer.
`*retBlk`	Pointer to the returned memory block.

Return Codes

`P1394_FS_OK`	Success.

Description

In **tsa1394EsbAllocFromPool**, the data buffer is given by the application.

tsa1394Freeb

```
extern tmLibappErr_t tsa1394Freeb(
   Int                instance,
   tsa1394MBlock_t   *bp
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`bp`	Pointer to the single mblock to be freed.

Return Codes

`P1394_FS_OK`	Success.

Description

Frees a message block and the associated data block if the reference count for the data block is 0. Otherwise, it frees only the message block. **bp** points to the block which has to be freed. The message block and associated data block are freed into the pool specified in **bp->lPoolId** and **bp->pData->lPoolId**. If the mblock was obtained using **tsa1394Esb-Alloc** or **tsa1394EsbAllocFromPool**, the associated **bp->pData->DbFrtn** is called.

tsa1394FreeMsg

```
extern tmLibappErr_t tsa1394FreeMsg(
   Int                 instance,
   tsa1394MBlock_t   *mp
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`mp`	Pointer to the chained mblocks to be freed.

Return Codes

P1394_FS_OK	Success.

Description

Frees the entire message with all its associated data buffers. **mp** points to the block which has to be freed

For each mblock within the message, the mblock and the associated data block are freed into respective **mblk->lPoolId**, **mblk->pData->lPoolId**, if the reference count, **pData->ucRefCount = 0**. If the data bock is freed and the mblock was allocated using **tsa1394EsbAlloc/tsa1394EsbAllocFromPool**, the application provided callback is called. If the reference count on the data block is not 0, only the mblock is freed.

Difference between tsa1394FreeMsg and tsa1394Freeb

Assume a logical message containing an mblock chain:

```
Mblk 1
pCont-----> Mblk 2
pData1  pCont--------> Mblk3
pData2  pCont-> Null.
pData3
```

tsa1394Freeb(Mblk1) This would free only **Mblk1** & its associated **pData1**.

tsa1394Freeb(Mblk2) This would free only **Mblk2** & its associated **pData2**.

whereas

tsa1394FreeMsg (Mblk1) This would try to free **Mblk1**, **Mblk2**, and **Mblk3**.

tsa1394FreeMsg This would follow through **pCont** and get to the next mblock and free that mblock as well.

tsa1394DupB

```
extern tmLibappErr_t tsa1394Dupb(
   Int                instance,
   tsa1394MBlock_t    *bp,
   tsa1394MBlock_t   **retBlk
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`bp`	Poiner to the single mblock to be duplicated.
`*retBlk`	Pointer to the returned memory block.

Return Codes

`P1394_FS_OK`	Success.

Description

Duplicates the message block. It creates a new message block header, sets the data block pointer to the old message block's data block, and increments the count in the data block. Input is a pointer to the block to be duplicated.

tsa1394DupMsg

```
extern tmLibappErr_t tsa1394DupMsg(
   Int                 instance,
   tsa1394MBlock_t    *mp,
   tsa1394MBlock_t   **retBlk
);
```

Parameters

`instance`	1394 instance, as returned by **tsa1394Open**.
`mp`	Poiner to the chained mblocks to be duplicated.
`retBlk`	Pointer to a pointer to the returned memory block.

Return Codes

`P1394_FS_OK`	Success.

Description

Duplicates the entire message. Input is a pointer to the message to be duplicated This results in a duplicate copy of an entire mblock chain within a single message, with each data area within each mblock of the message pointing to the same as the corresponding mblock in the original message.

1394 Callback Functions

Name	Page
tsa1394AllocCbFp_t	202
tsa1394FreeCbFp_t	202
tsa1394AddrReqCbFp_t	203
tsa1394AtReqCbFp_t	204
tsa1394SbEventCbFp_t	205
tsa1394CtrlEventCbFp_t	206

tsa1394AllocCbFp_t

```
typedef void *(*tsa1394AllocCbFp_t)(
   UInt32        size
);
```

Parameters

`size`	Size in bytes.

Return

`void*`	Pointer to a new buffer.

Description

The function is called to allocate a cache-enabled memory pool. It is registered when set up.

tsa1394FreeCbFp_t

```
typedef void (*tsa1394FreeCbFp_t) (
   void   *buf,
   void   *userdata
);
```

Parameters

`buf`	Pointer to be freed.
`userdata`	Any application data, if provided.

Return Codes

`void`	No return codes.

Description

The function is called to free a cache-enabled memory pool. It is registered when set up.

tsa1394AddrReqCbFp_t

```
typedef void *(*tsa1394AddrReqCbFp_t)(
   Int                   ach,
   tsa1394trDataInd_t   *ind,
   void                 *data
);
```

Parameters

`ach`	Indicates the async channel handle of the application which registered this callback function.
`ind`	Pointer to **tsa1394trDataInd_t** (data indication).
`data`	Pointer to application data.

Return Codes

`void`	No return codes.

Description

Function to be called with user data when incoming requests match the registered address space and transaction type. Registered by **tsa1394AsyncRegisterAddressSpace**.

tsa1394AtReqCbFp_t

```
typedef void (*tsa1394AtReqCbFp_t)(
   Int                   ach,
   tsa1394trDataCnfm_t  *conf,
   void                 *data
);
```

Parameters

`ach`	Indicates the Async channel handle of the application which registered this callback function.
`conf`	Pointer to tsa1394trDataCnFm_t (data confirmation info).
`data`	Pointer to application data

Return Codes

`void`	No return codes.

Description

Specifies the function pointer which is called with the response received from the remote node to this request. Registered by **tsa1394AsyncSendRequest**.

tsa1394SbEventCbFp_t

```
typedef void (*tsa1394SbEventCbFp_t)(
   Int                     ach,
   tsa1394SbEvent_t        busEvent,
   void                   *data,
   tsa1394SbEventInfo_t   *info
);
```

Parameters

`ach`	Indicates the Async channel handle of the application, which registered this callback function.
`busEvent`	Passed to the callback function indicating the bus event (as specified above) occurred.
`data`	Pointer to void application data.
`info`	Pointer to **tsa1394SbEventInfo_t** (not used currently).

Return Codes

`void`	No return codes.

Description

The application function called when serial bus event occurs. Registered by **tsa1394AsyncRegisterSbEvent**.

tsa1394CtrlEventCbFp_t

```
typedef void  (*tsa1394CtrlEventCbFp_t) (
   Int                        ach,
   tsa1394CtrlEvent_t         event,
   void                      *data,
   tsa1394CtrlEventInfo_t    *info
);
```

Parameters

`ach`	Indicates the Async channel handle which registered this callback function for control event specified as **controlEvent**.
`event`	Control Event.
`data`	Pointer to void application data.
`info`	Pointer to **tsa1394CtrlEventInfo_t** (not used currently).

Return Codes

`void`	No return codes.

Description

This function is called by the 1394 library when the control event occurs. Registered by **tsa1394AsyncRegisterControlEvent**.