Book 3—Software Architecture

Part B:

The Streaming Architecture

TriMedia

Version 2.1

Table of Contents

Chapter 7 TSSA Essentials

Chapter 8 Developing Applications Using a Streaming Model

Chapter 9 Applications Using a Non-Streaming Model

Chapter 10 TSSA Component Basics

Chapter 7

TSSA Essentials

Introduction

As TriMedia developers began creating large applications that handle multimedia data processing in real time, it became clear that some standardized methods would simplify the problem. The result is a set of guidelines collectively known as the TriMedia Streaming Software Architecture (TSSA). The sole mission of TSSA is to promote interoperability and reusability of components, thereby enabling a seamless collaboration between application programmers and component developers.

Many software issues have been encountered during the construction of complex TriMedia applications. The TSSA guidelines that have evolved from this experience will save you immense time in solving basic programming tasks.

What is TSSA?

TSA and TSSA have several features. Some may be useful for a given application, some may not. TSSA brings in all aspects of the TriMedia software architecture. It describes a method of constructing and connecting autonomous, task-based components that stream data between them.

TSSA provides a framework for components, whether streaming or not, that includes:

- A standard Application Programmer Interface (API)
- Common data formats (as defined in header files that are contained on the CD)

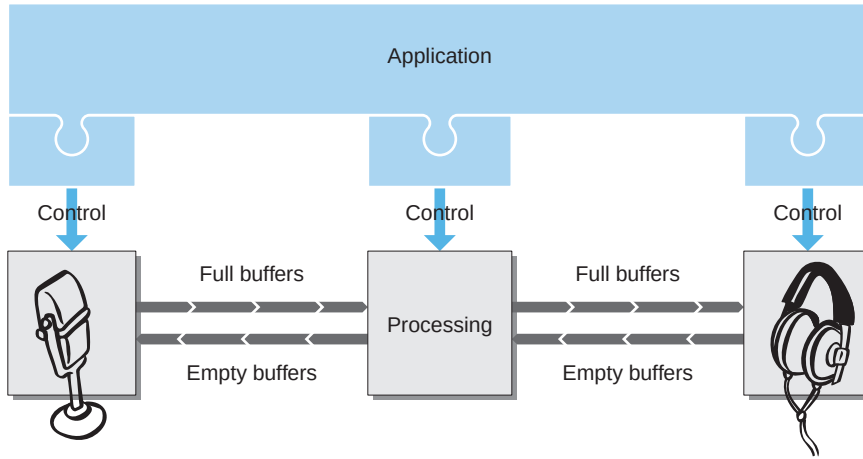


Figure 1 A TSSA-compliant component takes many shapes and sizes, but its entry points and the data formats flowing in and out are consistent and predictable.

Standardized APIs

Every library has an API reference that shares a consistent recognizable format. For instance, there is a page outlining each function and each data structure, and there is an overview that contains a picture of the inputs and outputs for that library.

References for TriMedia-supplied components are located in Book 5, *Device Library APIs*, and Book 6, *Software Library APIs* of the TriMedia SDE. If you are reading this page in its electronic pdf format, you can navigate to the API references via the bookmarks on the left side of the window.

Thoughts From the Architect

The following is an excerpt from an informal talk given by a key member of the original TSA design team. This excerpt may lend a personal context to the motivations that have led the development of the architecture.

> *I'd like to just go over some of the original thoughts we had early on. There were a number of things that we were trying to do, one of which was to have a streaming architecture, because it's a good way to start working on multimedia applications, where data is flowing through your system.*
>
> *One of the other goals was to create some consistency in our APIs, which we did not have. We wanted to get everybody's noses pointed in the same direction. There were people working on various pieces and we did not have a common view of how we did that. That led to divergence and since matters were not clear to ourselves, they were not clear to our customers.*
>
> *The device libraries are very similar with a number of functions that you'll see everywhere to control the devices on TriMedia. The idea was to build a number of layers on top of that which had a very similar API, and a very similar way of interacting. So far, we've dubbed those layers Application Libraries and OS layers, AL and OL for short. Those are now becoming more common terms.*
>
> *In these layers, you see a piece of software that does a certain function as a component, a black box component that has data streaming in and data streaming out. It doesn't have any global awareness. The only interfaces are data streaming in and data streaming out. Although there may be multiples of these, at some point, we didn't really start out that way.*
>
> *February 1998*

Common Data Structures

Given countless ways to describe data, TSA has set some rules on formats of data used among TSA components, so that the data is understandable to all. The standard TSA data structures are declared in a header file called tmAvFormats.h. (Refer to Chapter 4, *tmAvFormats.h: Multimedia Format Definitions* for documentation of these structures). This chapter gives you an introduction to these commonly used data structures.

The TSA data structures are used in communication among components, and between an application and its components. Between an application and its components, *configuration structures* set up components. Among components, *data packets* transfer data.

Data Packets

Data travels in packets between components. Components that produce data packets are referred to as "senders" and components that consume data are called "receivers." The packets are always constructed according to the TSA data packet structure, with a pointer to the packet header as its first field. The header contains descriptive information of the type of data in the packet. The packet has a flexible, yet consistent structure. Specific hooks are provided where application needs can be satisfied, provided that you adhere to a few TSA rules.

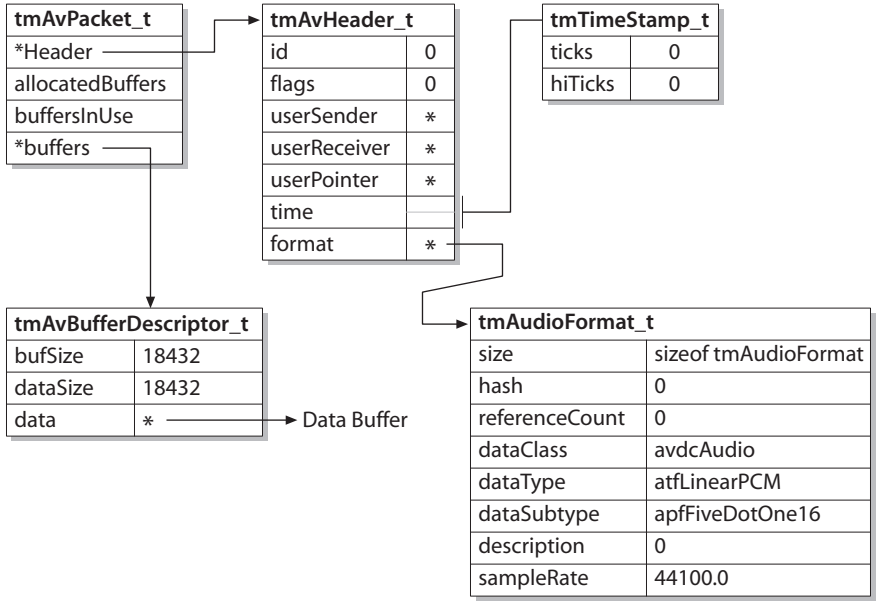


Figure 2 Example showing the hierarchical composition of a TSA packet

The Packet Structure (tmAvPacket_t)

The following shows he packet structure **tmAvPacket_t**:

```
typedef struct tmAvPacket_t {
   ptmAvHeader_t           header;
   UInt16                  allocatedBuffers;
   UInt16                  buffersInUse;
   tmAVBufferDescriptor_t  buffers[1];
} tmAvPacket_t, *ptmAvPacket_t;
```

The packet structure is declared in tmAvFormats.h, and more information is contained in the documentation of that file. Receiving components can count on finding a pointer to the header at the top of the packet. The last three fields combine to provide two services: the enabling of multiple data buffers, and the reporting of packet status as it moves from component to component.

The Header Structure (tmAvHeader_t)

The following shows he header structure **tmAvHeader_t**. Since the TSA default functions rely on the header, it must be allocated in every case. The function **tsaDefaultInOutDescriptorCreate** provides a standard mechanism to do this:

```
typedef struct tmAvHeader_t {
   UInt32          id;           /* read only to receiver          */
   UInt32          flags;        /* read only to receiver          */
   Pointer         userSender;   /* reserved for a user sender.
                                    read only to receiver.         */
   Pointer         userReceiver; /* reserved for a user receiver.
                                    read only to sender.           */
   Pointer         userPointer;  /* reserved for a user.
                                    read/write on both ends.       */
   tmTimeStamp_t   time;         /* time stamp                     */
   Pointer         format;       /* subclass of tmAvFormat_t       */
} tmAvHeader_t, *ptmAvHeader_t;
```

Provision is made in the header for application-specific data. More information will be found in the reference describing tmAvFormats.h

The timestamp field allows data packets to be easily time stamped. When the time stamp is valid, the packet is marked with a value in the flag field.

The Buffer Structure (tmAvBufferDescriptor_t)

```
typedef struct tmAvBufferDescriptor_t {
   UInt32    bufSize;
   UInt32    dataSize;
   Pointer   data;
} tmAvBufferDescriptor_t, ptmAvBufferDescriptor_t;
```

Like the rest of the data packet structures, the buffer structure is defined in tmAvFormats.h, and it is documented in Chapter 4, *tmAvFormats.h: Multimedia Format Definitions*. Packets can have any number of buffers. It is common for packets to have one buffer, but video packets, for example, often have three packets, for YUV data. Attaching numerous

buffers to one packet can be useful when the packets are small, as task switching overhead can be reduced.

Allocating a Packet

Packets with only one data buffer can be allocated statically, because one data buffer is provided by the packet structure by default. Packets with more than one data buffers must be allocated dynamically with malloc. Here is a way to allocate a packet with N data buffers. For YUV video packets, N would equal to 3, with one buffer for each of Y, U, and V.

```
ptmAvPacket_t ppacket = (ptmAvPacket_t) malloc(sizeof(tmAvPacket_t)
                      + (N-1) * sizeof(tmAvBufferDescriptor_t));
```

Format Structures (e.g. tmAvFormat_t)

Format structures describe the format of data. When used in the capabilities struct, they enumerate the formats supported by a component. The format manager checks two components for compatibility using the format in the capabilities struct. When used in the instanceSetup struct, a format structure allows the application to select the desired mode of operation. The generic form of the format struct is as follows:

```
typedef struct tmAvFormat_t {
   UInt32             size;
   UInt16             hash;
   UInt16             referenceCount;
   tmAvDataClass_t    dataClass;
   UInt32             dataType;
   UInt32             dataSubtype;
   UInt32             description;
} tmAvFormat_t, *ptmAvFormat_t;
```

The size field is required because there are different kinds of formats with different sizes. This field allows code that handles formats to get the size right. Beware of bugs caused by a failure to initialize the size field. The **hash** and **referenceCount** fields are used internally by the TSA format manager. The format manager ensures that the formats attached to packets are unique and valid. It enforces a safe discipline in the correct use of packets and formats.

The **dataClass**, **dataType**, and **dataSubtype** classify data types. Each of these fields is defined as an enumerated type with values equal to powers of two. This allows data types and subtypes to be ORed together when enumerating the capabilities of a component. And it allows a format request to be checked for validity with a simple AND of the request and the capabilities. Several example formats are described in the tmAvFormats reference. Note that the type "none" is not zero, but a specifically defined type. This allows the type or subtype "none" to be acceptable. Similarly, the type "generic" is defined as **0xFFFFFFFF**. This is used in a capabilities structure to indicate the acceptance of any type of data.

Time Structures (tmTimeStamp_t)

```
typedef struct tmTimeStamp_t {
    UInt32      ticks;
    UInt32      hiTicks;
} tmTimeStamp_t, *ptmTimeStamp_t;
```

Streaming data often needs a time stamp field (such as an MPEG presentation time stamp or video SMPTE time stamp). This field is always a part of the header. The type of the time stamp is maintained as an attribute of the stream. It is stored as the clock handle given to the components that handle this data.

Configuration Structures

The next set of structures is defined in tsa.h. While structures in tmAvFormats.h were available to and used by portions of the system below TSSA, the structures defined in tsa.h are specific to the streaming architecture.

Configuration structures set up components. In the simplest TSA application, **Capabilities**, **InstanceSetup**, and **IODescriptor** and **IODescriptorSetup** structures are used. While the Capabilities and InstanceSetup structures describe each component, **InOutDescriptor** and **InOutDescriptorSetup** describe the connection between components.

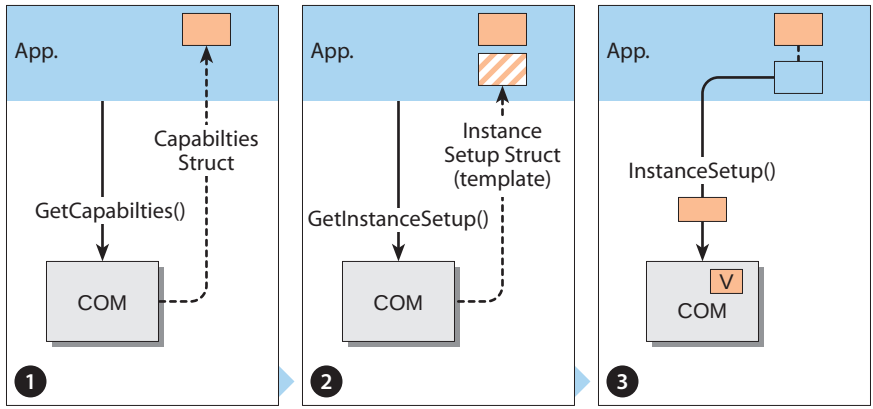


Figure 3 Typical behavior of configuration structures.

The example in Figure 3, depicts a typical start up sequence by the application to connect and run two components.

The application calls **getCapabilities** to get the capabilities structs ( ) from each of two component to be connected, then calls **Open** to open each component. It then passes the capabilities structs to a function, tsaInOutDescriptorCreate, via a InOutDescriptorSetup struct. The function returns an initialized InOutDescriptor struct describing the connection between the components.

Then, the application calls **getInstanceSetup** to get an instance setup struct template () containing initialized defaults previously set by **Open**. The application then fills in the template and passes the modified instance setup struct back to the component via an **InstanceSetup** call. The component stores this information internally in an instance variable (V).

Note
A detailed example further explains the sequence in Chapter 8.

Capabilities Structures

The capabilities struct of a component is in the following form, with the default capabilities struct as the first field.

```
typedef struct tsaComCapabilities {
   ptsaDefaultCapabilities_t defaultCapabilities;

   /* component specific extension */
   ...
} tsaComCapabilities_t, *ptsaComCapabilities_t;
```

tsaDefaultCapabilities_t

This default capabilities struct includes fields for the component class and version number, the resource requirements of the component, a description of all inputs and outputs of the component, among other fields.

```
typedef struct tsaDefaultCapabilities {
   tmComponentClass_t             componentClass;
   tmVersion_t                    version;
   UInt32                         capabilityFlags;
   Int                            textmemoryRequirement;
   Int                            datamemoryRequirement;
   Int                            processorRequirement;
   UInt                           numSupportedInstances;
   UInt                           numCurrentInstances;
   UInt                           numberOfInputs;
   ptmAvFormat_t                 *inputFormats;
   UInt                           numberOfOutputs;
   ptmAvFormat_t                 *outputFormats;
   tsaReceiverFormatSetupFunc_t   receiverFormatSetup;
} tsaDefaultCapabilities_t, *ptsaDefaultCapabilities_t;
```

The capabilities struct of a component describes the capabilities of a component. The default capabilities struct describes numerous attributes that most components have. All components have a componentClass, a version, capabilities flags, resource requirements, and the numbers of supported and current instances. The fields concerning inputs/outputs and formats apply to all TSSA (streaming) components, and are used by the format manager to validate streams. The fields describing memory and processor usage are not yet used.

tsaInOutDescriptor_t

An InOutDescriptor describes the connection between two streaming components. One instance of this structure is shared by the two connected components. The application usually creates this connection structure by calling **tsaDefaultInOutDescriptorCreate**. The internals of the **InOutDescriptor** structure are meant to be private.

When an InOutDescriptor is no longer being used (i.e., the application had closed both components), the application then must call tsaDefaultInOutDescriptorDestroy on the InOutDescriptor previously created with its counterpart.

tsaInOutDescriptorSetup_t

An InOutDescriptorSetup_t gives you access to the public fields:

```
typedef struct tsaInOutDescriptorSetup {
   ptmAvFormat_t                 format;
   tsaInOutDescSetupFlags_t      flags;
   String                        fullQName;
   String                        emptyQName;
   UInt32                        queueFlags;
   ptsaDefaultCapabilities_t     senderCap;
   ptsaDefaultCapabilities_t     receiverCap;
   UInt32                        senderIndex;
   UInt32                        receiverIndex;
   UInt32                        packetBase;
   UInt32                        numberOfPackets;
   UInt32                        numberOfBuffers;
   UInt32                        bufSize[1];
} tsaInOutDescriptorSetup_t, *ptsaInOutDescriptorSetup_t;
```

Detailed descriptions of each field are found in the documentation describing tsa.h.

Instance Setup Structures

The instance setup struct of a component is in the following form, with the default instance setup struct as the first field.

```
typedef struct tmalComInstanceSetup {
   ptsaDefaultInstanceSetup_t   defaultInstanceSetup;

   /* component specific extension */
   ...                              ...
} tmalComInstanceSetup_t, *ptmalComInstanceSetup_t;
```

tsaDefaultInstanceSetup_t

While all components have an instance setup struct, the default instance setup structure is applicable only to TSSA components. The default instance setup structure contains all the information necessary to set up a TSSA component. It contains, among other fields,

pointers to TSSA callback functions, descriptors for all inputs and outputs, a clock handle, and a parent ID, as shown here.

```
typedef struct tsaDefaultInstanceSetup {
   Int                           qualityLevel;
   tsaErrorFunc_t                errorFunc;
   UInt32                        progressReportFlags;
   tsaProgressFunc_t             progressFunc;
   tsaCompletionFunc_t           completionFunc;
   tsaDatainFunc_t               datainFunc;
   tsaDataoutFunc_t              dataoutFunc;
   tsaMemallocFunc_t             memallocFunc;
   tsaMemfreeFunc_t              memfreeFunc;
   ptsaClockHandle_t             clockHandle;
   ptsaInOutDescriptor_t         inputDescriptors;
   ptsaInOutDescriptor_t         outputDescriptors;
   UInt32                        parentId;
   tsaControlFunc_t              controlFunc;
   ptsaControlDescriptor_t       controlDescriptor;
   UInt32                        priorit;
} tsaDefaultInstanceSetup_t, *ptsaDefaultInstanceSetup_t;
```

Callback functions are provided for applications to control to a certain degree the operations of a component. They are a critical part of TSA and described in depth beginning in *Callback Functions* on page 28.

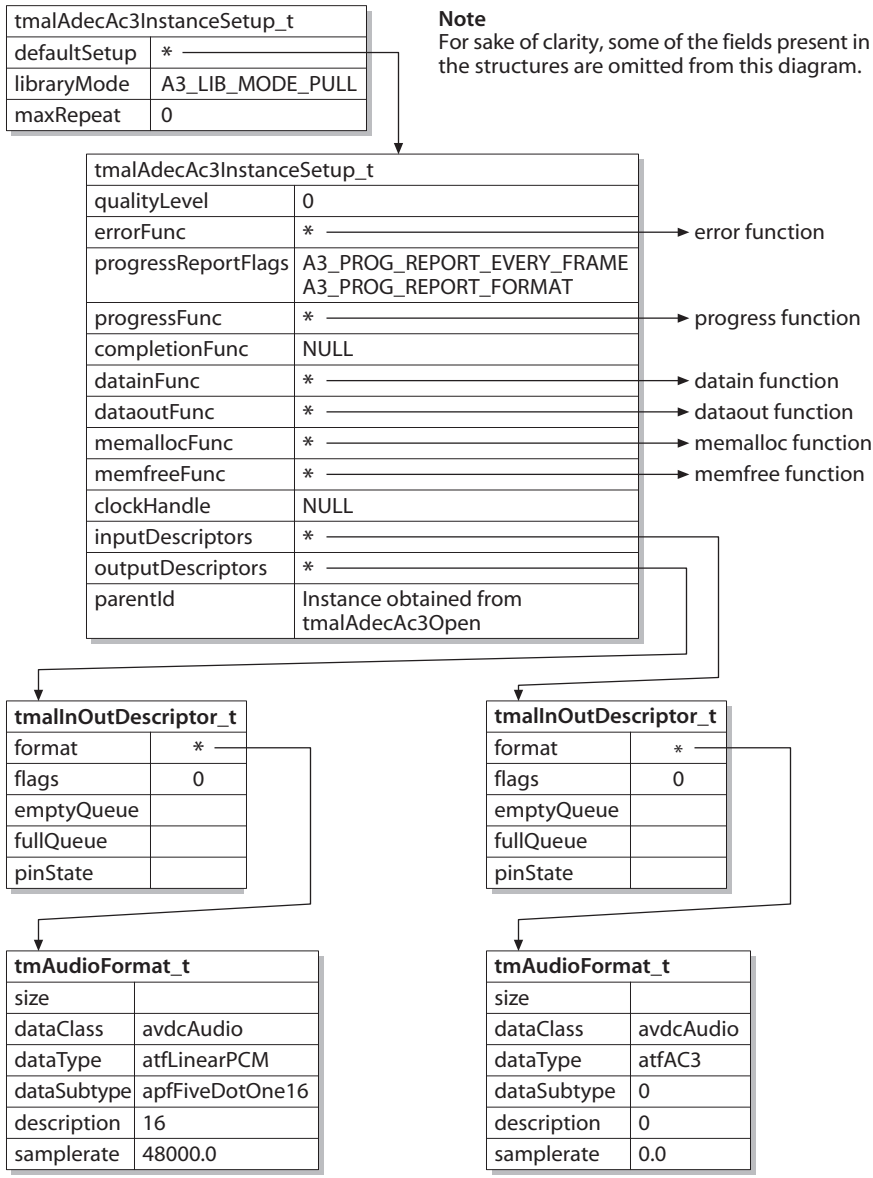


Figure 4 Hierarchy of an instance setup structure (this example is for the Dolby Digital AC-3 decoder)

Types of TSSA Component Interfaces

Several distinct interface types are applied to TSSA components. These interfaces are sometimes called the "AL" and "OL" layers. A similar distinction is sometimes made between a "streaming" and "non-streaming" interface, although the two are not strictly equivalent.

- Asynchronous (sometimes referred to as *streaming*, *pull*, or *data-driven*) components are components that run in their own context separate from the application. Asynchronicity is generally (though not always) accomplished by means of an operating system such as pSOS™, and exists only at what TriMedia calls the Operating System Layer (OL).

  Chapter 8, *Developing Applications Using a Streaming Model*, describes in more detail the process of developing applications with asynchronous components.

- Synchronous (sometimes referred to as *non-streaming*, *push*, or *control-driven*) components are components that run within the application's context. In general, these components work without an operating system. This is more like a traditional library model (i.e., a procedural or functional interface).

  Chapter 9, *Applications Using a Non-Streaming Model*, describes in more detail the process of developing applications with synchronous components.

Note
Chapter 10, *TSSA Component Basics*, describes the process of developing TSA-compliant components.

Asynchronous Components (TSSA)

In multimedia systems, the ability for applications to switch tasks while other components process data in their own context is essential. This is a natural model for the encoding and decoding of audio and video, for example, where signals are processed continuously. TSSA supports this data-driven model with its standard data types and its default behaviors (default functions).

TSSA components are pieces of software, whose functionality is accessible through a predictable set of entry points. Components have no global awareness. They are self-contained modules to be used as building blocks for applications. The application uses a common API to set up a component and to connect it to other components. It then starts the component, at which point the component begins to run within its own context, either in a separate task or ISR. Data access is handled inside asynchronous components. At its most basic operation, a component gets a data packet, processes the data, and passes it on. The packets (and their associated memory buffers) are recycled by placing them in queues of empty packets.

Synchronous Components

Some applications may behave more in a control-driven way. In this case, the library is used in the context of the application (i.e., the component is accessed through function calls and does not have a separate context). The timing of the processing is completely determined by the application. Examples of synchronous components are the 2D graphics library and the window manager.

In addition, some components that are normally used asynchronously also support synchronous implementation. This is further explained in Chapter 8, under the ProcessData function. The Dolby Digital decoder can also be operated in a synchronous mode.

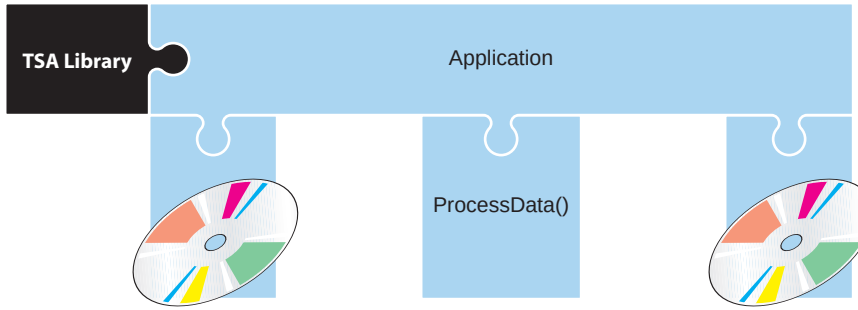


Figure 5 Synchronous components operate in the context of the application

When using synchronous components, data access must be handled in the application and passed on to the components as function parameters. While this may be a detail that the application programmer would not like to see, it is an option that provides some flexibility in data access.

The AL layer and the OL layer

It is common for the asynchronous interface to be implemented at the OL layer, while the synchronous interface is implemented at the AL layer. The key factor distinguishing the AL layer from the OL layer is the use of an operating system at the OL layer to support multi-tasking. In fact, it has been suggested that "multi-tasking layer" would be a better name than "Operating System Layer." Historically, this distinction was introduced to separate the types of control code that rely on operating system specific features from the code that implements the highly optimized signal processing making up the core of multimedia codecs. And for the implementation of codecs, this works very well. But there are times when little is gained by separating code between the AL and OL layers. In these cases, the OL layer interface is used exclusively and the AL layer interface may atrophy. You might consider that the OL layer interface is the generally required public API, and the AL layer interface is an implementation detail. Splitting your component into AL and OL layers makes it easy for you to re-use the default TSSA features.

Audio Signal Processing (ASP) Components

A special class of TSA compliant components are known as ASP (Audio Signal Processing) components. ASP components are very similar to AL layer TSSA components. They share the standard **Open**, **Close**, **InstanceSetup**, and **InstanceConfig** functions that are used with all TSA modules. But ASP components do not need to provide **Start** and **Stop** functions. ASP components are asynchronous. ASP components export the **processData** function that is normally optional in an AL layer component. The format of the data buffer used by ASP components is also standardized, and it is driven by the requirements of audio mixers. The **tsaAspChannelBuffer_t** describes the data to be processed. This buffer structure supports circular buffering, and it standardizes the data itself to be 32-bit integers.

It is easy to convert an ASP component into an AL layer component, and by extension, to an OL layer component. If the instance variables of an ASP component place pointers to the default structures in their standard locations, they can be filled in later. When a component is used only at its ASP layer, then these pointers can be Null.

TSSA Overview

Component Classes

The asynchronous TSSA components exist in one of three major classes: digitizers, processors (also known as filters), and renderers. Digitizers have only outputs. Renderers have only inputs. Filters have both outputs and inputs.

Each filter component has a task as its context, while renderers and digitizers are interrupt-based, because they need signals from hardware in their operations.

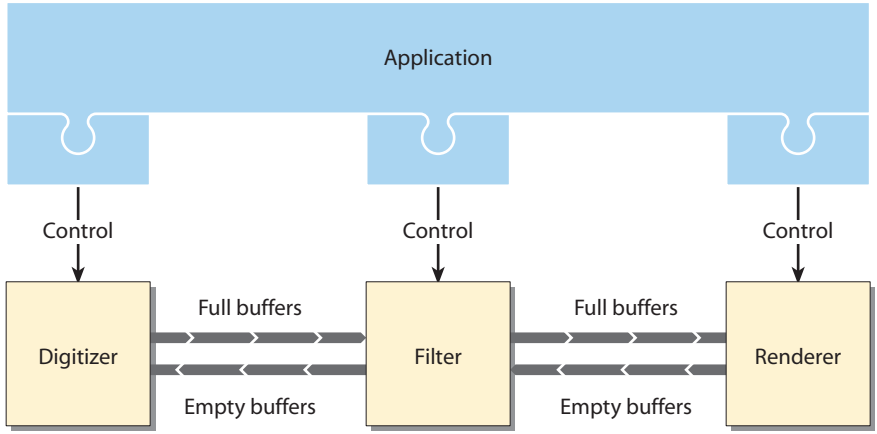


Figure 6 Classes of streaming components. This example illustrates an audio processing application.

Component Connections: Queues and Data Packets

The inputs and outputs of TSSA components are embodied in queues and the data packets populating the queues. TSSA components are data driven—a component is scheduled based on the availability of data through its queues.

Data transmission within TSSA applications always occurs in the form of data packets sent between components, as discussed in *Data Packets* starting on page 14. Every connection between the output of one component and the input of another consists of at least two data queues. One queue carries full data packets from the sender to the receiver and the other returns empty packets to the sender to recycle packet memory.

The empty packets are returned to signal that the data has been received properly and that the memory associated with the data packet may be reused. This "recycling" of memory relieves the nuisance of always having to free and reallocate memory, and prevents memory leaks in components.

Additionally, the choice of packet size and the number of packets influence the behavior of the system. For example, the use of smaller packets will increase the number of task switches in the system because most components process one buffer at a time. While the use of larger packets will result in fewer task switches, it can also result in longer latency as receiving components wait for packets to be filled. These and other issues (such as cache behavior) must be balanced to produce the desired result.

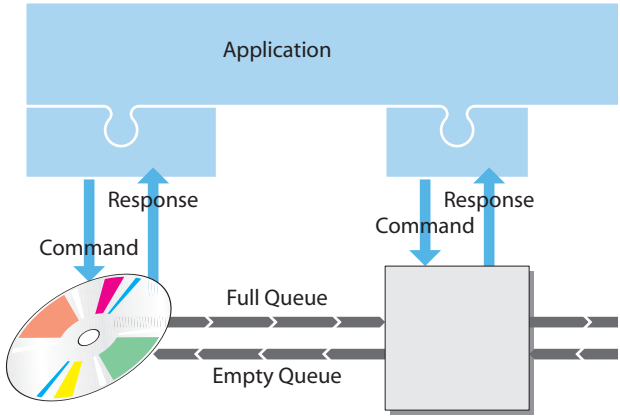


Figure 7 Queues connecting TSSA components

Function API

All TSSA components contain some of the following entry points shown in Table 1.

Table 1 TSSA API Functions

Function	Description
tmolComGetCapabilities	Requests the component capabilities, including the component class, the number of supported instances, the version number, resource requirements, the number of inputs/outputs, and supported data formats on these inputs/outputs.
tmolComOpen	Requests an instance of the component. Components that use a specific hardware device usually allow only one instance. Full software components usually allow multiple (sometimes unlimited) instances. Memory for instance variables is allocated in the Open call.
tmolComClose	Releases the component instance. Frees memory.
tmolComGetInstanceSetup	Retrieves a pointer to an accurate instance setup structure. When the component is created, this structure is filled with default values.

Table 1 TSSA API Functions

Function	Description
tmolComInstanceSetup	Sets up the component instance. All necessary information is passed through a structure pointer.
tmolComStart	Starts the data processing of the component instance.
tmolComStop	Stops the data processing of the component instance.
tmolComInstanceConfig	Changes the behavior of the component while it is running. Any type of setting like channel, volume, or output mode should be set using this function. This function operates on control queues connecting application to component. Note that this function provides a bidirectional interface to the component. Not only can it be used to change settings of the running component, but also for obtaining information from the component.
tmolComStopPin	Stop data access from or to a specific pin (connection)
tmolComUnStopPin	UnStop (restart) data access from or to a specific pin

For most components, the previous functions appear in both OL and AL layers. The OL layer functions are prefixed with "tmol", and the AL layer functions are prefixed with "tmal".

At the OL layer, all components operate asynchronously. Components achieve asynchronous operation by processing data either in a task or an ISR. Renderers and Digitizers are components that interface with peripheral devices. Since interrupt service routines (ISRs) are usually required to communicate with the peripheral devices, it is common for renderers and digitizers to do all of their processing in the ISR. Otherwise, most components will have one or more operating system tasks (threads) that process data.

All compliant TSA components implement functions that return a value of type **tmLibappErr_t.** Other results are delivered via pointer parameters. The include file, tmLibappErr.h, lists the different errors currently possible, either default operations or components specific operations. Component specific error codes must be created with a component to avoid having the same error as another component. An error base must be defined in accordance with the rules outlined in tmLibappErr.h.

It is in the OL layer that dependencies on operating systems and the details of streaming communications are determined. In the TriMedia implementation of the architecture, the pSOS operating system is utilized through the Operating System Abstraction Layer (OSAL). This layer was created to simplify the task of porting to another operating system. Because most of the functionality of OL layers are similar, TriMedia provides a default layer to handle most of the component's interaction with the OS. With a default layer included, the creation of an OL layer is a very simple extension of a working AL layer.

Callback Functions

To achieve OS-independence, the AL layer of a TSSA component uses the defined set of callback functions shown in Table 2. This set of callback functions include data access, event reporting, and memory management functions.

Table 2 TSSA Callback Functions

Function	Description
datainFunc	Requests a block of data on one of the inputs of the component. The function is called to obtain a new full buffer of data to be processed, or to return a previously obtained buffer on which processing is complete and is now considered to be empty.
dataoutFunc	Delivers a block of processed data on one of the outputs of the component. The function is called either to obtain an empty buffer in which processed data can be delivered, or to deliver a previously obtained empty buffer that now contains processed data and is now thus considered full.
memallocFunc	Allocates a block of memory.
memfreeFunc	Frees a previously allocated block of memory.
controlFunc	Requests or acknowledges a control message.
errorFunc	Reports an error encountered during the processing of the data.
progressFunc	Reports that a certain amount of progress has been made on the data processing.
completionFunc	Reports completion of the data processing.

While the application programmer is encouraged to provide suitable error, progress, and completion functions, the other callbacks are part of the default implementation which handles most of TSSA and, in most cases, should not be overwritten.

For more information on default callback functions, see Chapter 10, *TSSA Component Basics*.

Chapter 8

Developing Applications Using a Streaming Model

Sample Application

The best way to understand how the streaming model works is to study a sample application using components in the TSSA OL layer. This chapter contains pseudo code from a basic TSSA application that uses three generic asynchronous components: a digitizer, a processor, and a renderer. After the basic application has been explained in its entirety, more advanced issues of TSSA applications will be presented starting on page 41.

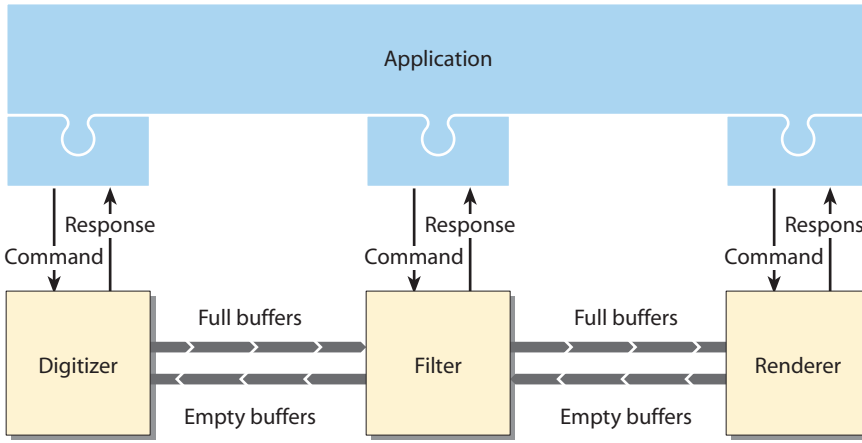


Figure 8 Overall flow of the sample application showing a command/response queue between application and component and bidirectional data queues between components

Following is the pseudo code of the sample application. Each part of the code will explained on the following pages. Although error checking is critical in any TSSA application, it is omitted from this example for clarity.

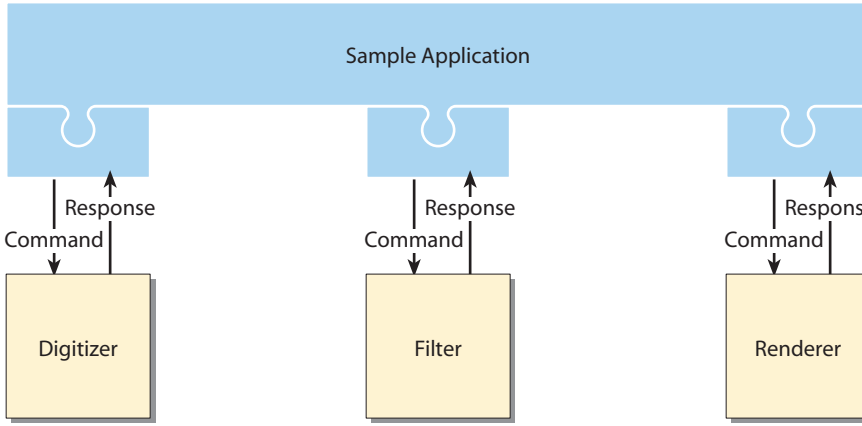
```
void tmosMain(){
   Int                            digInst;
   Int                            procInst;
   Int                            rendInst;
   ptmolDigitizerInstanceSetup_t digSetup;
   ptmolProcessorInstanceSetup_t procSetup;
   ptmolRendererInstanceSetup_t  rendSetup;
   ptmolDigitizerCapabilities_t  digCap;
   ptmolProcessorCapabilities_t  procCap;
   ptmolRendererCapabillities_t  rendCap;
   ptsaInOutDescriptor_t         iodesc1;
   ptsaInOutDescriptor_t         iodesc2;
   ptsaInOutDescriptorSetup_t    iosetup;

   tmAvFormat_t lformat = {
      sizeof(tmAvFormat_t), /* size           */
      0,                    /* hash           */
      0,                    /* referenceCount */
      avdcGeneric,          /* dataClass      */
```



```
      avdtNone,             /* dataType       */
      avdsNone,             /* dataSubtype    */
      0,                    /* description    */
   };

/* Get Component Capabilities */
   tmolDigitizerGetCapabilities( &digCap  );
   tmolProcessGetCapabilities  ( &procCap );
   tmolRendererGetCapabilities ( &rendCap );

/* Open Components */
   tmolDigitizerOpen( &digInst  );
   tmolProcessOpen  ( &procInst );
   tmolRendererOpen ( &rendInst );

/* Create InOutDescriptors */
/* Set up InOutDescriptorSetup struct */
   iosetup.format          = &lformat;
   iosetup.flags           = 0;
   iosetup.fullQName       = "DPFQ";
   iosetup.emptyQName      = "DPEQ";
   iosetup.queueFlags      = tmosQueueFlagsStandard;
   iosetup.senderCap       = digCap->defaultCapabilities;
   iosetup.receiverCap     = procCap->defaultCapabilities;
   iosetup.senderIndex     = 0;
   iosetup.receiverIndex   = 0;
   iosetup.numberOfPackets = NUM_OF_PKTS;
   iosetup,numberOfBuffers = 1;
   iosetup.bufSize[0]      = DATASIZE;
   tsaDefaultInOutDescriptorCreate(iodesc1, &iosetup);

/* Set up InOutDescriptorSetup struct again */
   iosetup.format          = &lformat;
   iosetup.flags           = 0;
   iosetup.fullQName       = "PRFQ";
   iosetup.emptyQName      = "PREQ";
   iosetup.queueFlags      = tmosQueueFlagsStandard;
   iosetup.senderCap       = procCap->defaultCapabilities;
   iosetup.receiverCap     = rendCap->defaultCapabilities;
   iosetup.senderIndex     = 0;
   iosetup.receiverIndex   = 0;
   iosetup.numberOfPackets = NUM_OF_PKTS;
   iosetup,numberOfBuffers = 1;
   iosetup.bufSize[0]      = DATASIZE;
   tsaDefaultInOutDescriptorCreate(iodesc2, &iosetup);

/* Get instance setup structures */
   tmolDigitizerGetInstanceSetup( digInst,  &digSetup  );
   tmolProcessGetInstanceSetup  ( procInst, &procSetup ) ;
   tmolRendererGetInstanceSetup ( rendInst, &rendSetup );

   digSetup->outputDescriptors[0]  = iodesc1;
   procSetup->inputDescriptors[0]  = iodesc1;
   procSetup->outputDescriptors[0] = iodesc2;
   rendSetup->inputDescriptors[0]  = iodesc2;

/* Set up components */
   tmolDigitizerInstanceSetup( digInstance, digSetup   );
   tmolProcessInstanceSetup  ( procInstance, procSetup );
   tmolRendererInstanceSetup ( rendInstance, rendSetup );

/* Start processing */
```

```
    tmolDigitizerStart( digInst  );
    tmolProcessStart  ( procInst );
    tmolRendererStart ( rendInst );

/* Wait for a stop condition */
    while( running ){}

/* Stop components */
    tmolRendererStop ( rendInst );
    tmolProcessStop  ( procInst );
    tmolDigitizerStop( digInst  );

/* Close instances */
    tmolRendererClose ( rendInst );
    tmolProcessClose  ( procInst );
    tmolDigitizerClose( digInst  );

/* Destroy InOutDescriptors */
    tsaDefaultInOutDescriptorDestroy( iodesc1 );
    tsaDefaultInOutDescriptorDestroy( iodesc2 );
}
```

Getting Component Capabilities




```
tmolDigitizerGetCapabilities(&digCap);
tmolProcessGetCapabilities(&procCap);
tmolRendererGetCapabilities(&rendCap);
```

The first application calls the **tmolComGetCapabilities** function of each component that will be used. It then extracts the default capabilities structures from the capabilities structures received from each of the components and passes each pair into the **tsaDefaultInOutDescriptorCreate** function. The formats in the default capabilities structures are used to guarantee that each pair of connected components is compatible.

Opening Components

```
tmolDigitizerOpen(&digInst);
tmolProcessOpen(&procInst);
tmolRendererOpen(&pendInst);
```

The application calls the **tmolComOpen** function of each component in turn. This will allocate the memory needed for the instance variables of each component. Remember to always check the return value.

Creating InOutDescriptors

```
iosetup.format          = &lformat;
iosetup.flags           = 0;
iosetup.fullQName       = "DPFQ";
iosetup.emptyQName      = "DPEQ";
iosetup.queueFlags      = tmosQueueFlagsStandard;
iosetup.senderCap       = digCap->defaultCapabilities;
iosetup.receiverCap     = procCap->defaultCapabilities;
iosetup.senderIndex     = 0;
iosetup.receiverIndex   = 0;
iosetup.numberOfPackets = NUM_OF_PKTS;
iosetup.numberOfBuffers = 1;
iosetup.bufSize[0]      = DATASIZE;
tsaDefaultInOutDescriptorCreate(iodesc1, &iosetup);

iosetup.flags           = 0;
iosetup.fullQName       = "pRFQ";
iosetup.emptyQName      = "PREQ";
iosetup.queueFlags      = tmosQueueFlagsStandard;
iosetup.senderCap       = procCap->defaultCapabilities;
iosetup.receiverCap     = rendCap->defaultCapabilities;
iosetup.senderIndex     = 0;
iosetup.receiverIndex   = 0;
iosetup.numberOfPackets = NUM_OF_PKTS;
iosetup.numberOfBuffers = 1;
iosetup.bufSize[0]      = DATASIZE;
tsaDefaultInOutDescriptorCreate(iodesc2, &iosetup);
```

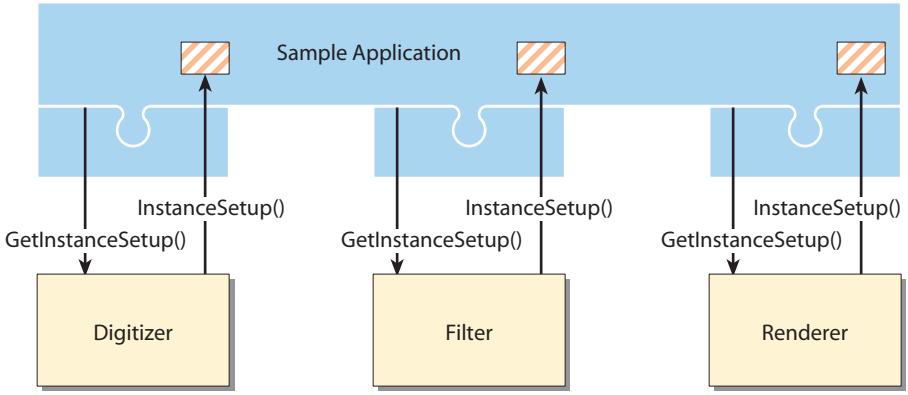
The connections between each pair of components are specified and described in a structure of the type **tsaDefaultInOutDescriptor_t**. This structure is created using the function **tsaDefaultInOutDescriptorCreate** according to the values in the corresponding setup structure of type **tsaDefaultInOutDescriptorSetup_t**.

The core of an **InOutDescriptor** is the pair of full and empty queues that buffer data packets and control scheduling. The setup structure can specify the names and the properties of the full and empty queues for debugging.

Another critical part of an **InOutDescriptor** is the format. When specified, it is checked in **tsaDefaultInOutDescriptorCreate** against the capabilities of the sender and the receiver. To distinguish among several pins of the sender and the receiver, **senderIndex** specifies the index in the output array of the sender and **receiverIndex** specifies the index in the input array of the receiver. *Delaying Setting of Format* starting on page 41 discusses the cases where the application does not specify the format.

Data is exchanged between two components using packets of the type **tmAvPacket_t**. Pointers to these packets are passed as messages in the full and empty queues. The **InOutDescriptorCreate** function creates packets to be used in the connection. The **InOutDescriptorSetup** structure can specify the number of packets, the number of buffers in each packet, and the size of each buffer of the packet in bytes. The **flags** field, when set to **tsaIODescSetupFlagCacheMalloc**, will cause the packets to be cache malloced and aligned. "Creating Packets Without InOutDescriptorCreate" starting on page 41, discusses the cases where the application needs the packets to be created in a special way.

Getting Instance Setup Structures




```
tmolDigitizerGetInstanceSetup(digInst, &digSetup);
tmolProcessGetInstanceSetup(procInst, &procSetup);
tmolRendererGetInstanceSetup(rendInst, &rendSetup);
```

Each component is set up according to a corresponding instance setup struct. The application gets a template instance setup struct of the component by calling the **tmolComGetInstanceSetup** of each component. In the function, the template instance setup struct is initialized with default values and returned to the application. Note that **tmolComGetInstanceSetup** can be used at any time. It will always return a pointer to the current instance setup structure.

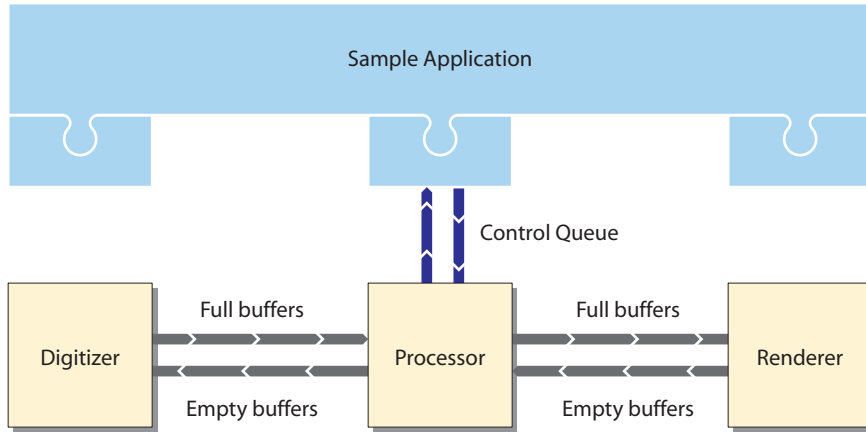
Sending in the InOutDescriptors to the Components

```
digSetup ->outputDescriptors[0] = iodesc1;
procSetup->inputDescriptors [0] = iodesc1;
procSetup->outputDescriptors[0] = iodesc2;
rendSetup->inputDescriptors [0] = iodesc2;
```

The components are connected by the **InOutDescriptor** structures sent in from the application. In the previous example, the digitizer's output index 0 is connected to the pro-

cessor's input index 0. Similarly, the processor's output index 0 is connected to the renderer's input index 0.

Setting Up Components




```
tmolDigitizerInstanceSetup( digInst,  digSetup  );
tmolProcessInstanceSetup  ( procInst, procSetup );
tmolRendererInstanceSetup ( rendInst, rendSetup );
```

After the instance setup structures are filled in as needed by the component, they are passed into the components via the **tmolComInstanceSetup** functions. Again, it is important to check the return value of this function.

Callback Functions

TSSA components rely on callback functions for significant parts of their functionality. Callback functions are used by the components to communicate with the application, while the application has full control over what the callback functions do. TSSA provides a number of callbacks. When these are provided by the application, they will be called back into the application from the context of the component. A number of callback functions are listed in the default instance setup structure. More details on these can be found in Chapter 10, *TSSA Component Basics*. This section describes the callback functions from the point of view of an application programmer.

The Progress Function

The usage of the progress function varies with each component. The progress function provides a flexible synchronization point between the application and the component. It is a way for the component writer to give the application some control at that point in the component's code. The component can have specific progress codes as inputs to the progress function. Some examples of component use of the progress function are:

- Components that decode compressed data streams are likely to use the function to inform the application of the format of the compressed stream, as this is often determined after some initial decoding.
- The file reader calls its progress function when the end of the file has been read to give the application a chance to stop it.
- The audio renderer calls its progress function in the interrupt service routine. This allows application-specific synchronization code to be called on each interrupt.

Note that progress functions might be required to be re-entrant. It is a good idea to assume that this is the case, although component documentation should state if the progress function is required to be re-entrant. This means that you should avoid the use of static or global variables in your progress function.

The Error Function

The error function is called by a component when it encounters an error. Flags specify whether the error is fatal. The error function gives the application writer an opportunity to insert an appropriate error handler.

Note
Error functions might be required to be re-entrant. It is a good idea to assume that this is the case, although component documentation should state if the error function is required to be re-entrant. This means that you should avoid the use of static or global variables in your error function.

The Completion Function

The completion function is called by task-based components when it is exiting its main loop and again when the stop sequence is finished. Since this loop is contained in the **tmolComStart** function, it is sometimes referred to as the start loop. Like the error function, it is a very specific form of a progress indication.

Memory Management Functions

Sometimes you may want to override the default memory manager, which uses TriMedia's **_cache_malloc** and **_cache_free**. When a component is opened, it is likely to request a small amount of memory for its instance variables using **malloc**. The exact size of this memory is specified in the capability structure. But components that require local memory buffers can be designed to get and free that memory using the memory callback functions. By default, these are patched to use the TriMedia's **_cache_malloc** and **_cache_free** function.

Best Practice:
TSSA components should avoid calls to malloc and free while running. This can lead to serious memory fragmentation issues. Instead, allocate all required memory in instance setup. If your component needs to allocate and free memory, use the memspace manager to localize this memory pool.

The Data Access Functions

datain and **dataout** should not be changed at the application level. As with each of the callback functions, leaving them Null in the instance setup structure will result in the installation of a default. The default **datain** and **dataout** functions are part of the TSSA core. We do not recommend creating completely compatible substitutes for the default **datain** and **dataout** functions.

Task Control

The default instance setup structure contains a number of fields that control the behavior of the component's task. These include the task priority, task flags, and the task name.

Priority

Since pSOS is a priority-based multi-tasking operating system, the relative priorities of a group of tasks is very important, and is intertwined into the working of a multitasking system.

IMPORTANT
Unless the component in question uses no tasks, you must fill in this variable.

You can discover whether a component uses a task by inspecting the **createNoTask** member of the default instance setup structure. Components such as renderers will set this to true, in which case the priority is ignored.

The assignment of priorities has been the topic of much research in computer science. It is assumed that the application programmer is familiar with these issues, and this variable sets the priority of tasks created by the component. Under pSOS, the lowest user priority is 1, and the highest is 239. Refer to the pSOS documentation on this CD for more information.

Task Flags

pSOS also allows tasks to be created with a number of properties, such as whether tasks can be preempted or are time-sliced. Otherwise, this field will be set to a recommended default in the component.

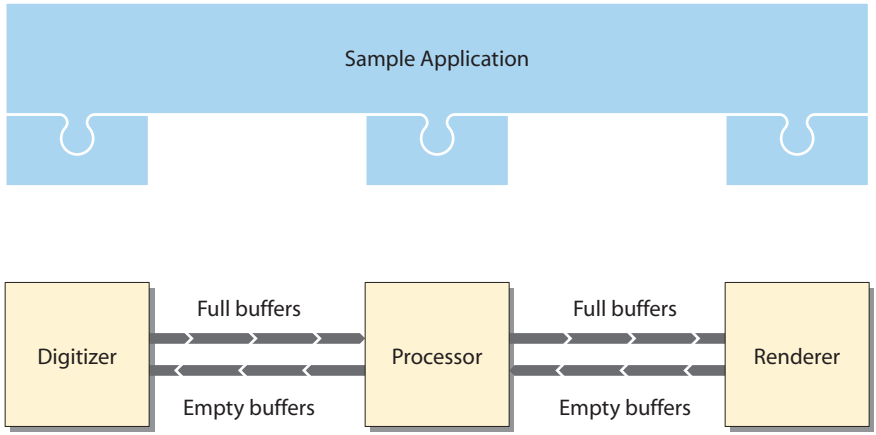
Task Name

The default instance setup structure reserves 16 characters for use as a task name. This field is mainly used for debugging. For pSOS, only the first four letters of the name will be used. When using pSOS, the tasks can be examined in **tmdbg** using the pSOS monitor. If there are multiple instances of a component, it is recommended that you use this field to identify tasks.

Stack Size

Although a default stack size is set for each component task, it can be overwritten with this field. Remember that ISRs use the stack of the task that they interrupt.

Start Processing

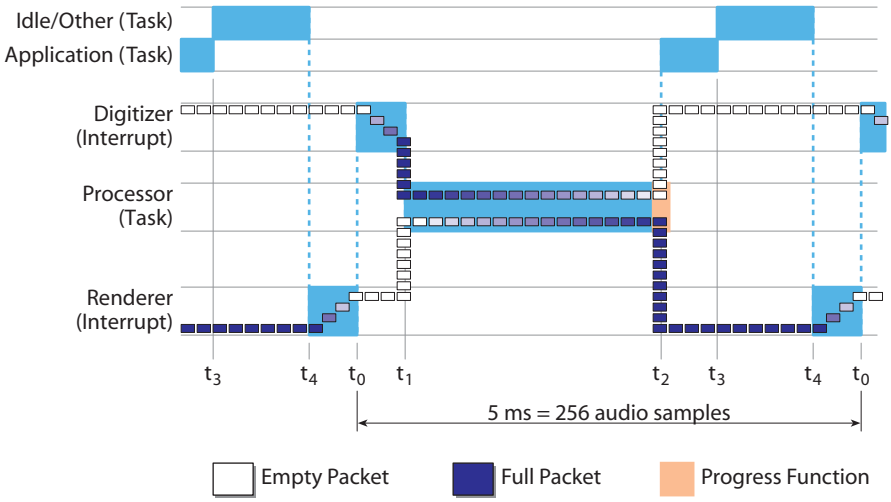


DATPULL7.MOV

```
tmolDigitizerStart( digInst  );
tmolProcessStart  ( procInst );
tmolRendererStart ( rendInst );
```

When an application calls **tmolComStart**, a separate thread of execution is created. If the component is task-based, as in most processor components, the thread runs in a task with its own stack. Alternatively, renderers and digitizers are interrupt-based and therefore run in an ISR while using stack space from the application.

The following is an example of the scheduling of a number of tasks in a TSSA audio application. The tasks involved are the root task, the idle task, the processor component task, and their interaction with two interrupt service routines (an audio digitizer and an audio renderer).



TIMELINE.MOV

Figure 9 Processor control flow in a TSSA system. The animation shows the dynamic interaction between components, packets, and the application.

Following are descriptions of the task switches illustrated in Figure 9:

Task Switch t_0	Audio input interrupt triggers the audio digitizer. Digitizer procures an empty packet, fills it, and sends it to the filter.
Task Switch t_1	The filter task has been waiting for a full packet. Since a pSOS call was made in the digitizer's ISR, the pSOS scheduler is invoked on return from the ISR. Control is transferred to the waiting filter task.
Task Switch t_2	The filter finishes processing. An input buffer has been consumed and an output buffer is produced. Control is transferred to the next waiting task, which is the application (root task).
Task Switch t_3	The application completes its processing and control is transferred to some other task.
Task Switch t_4	The audio renderer is triggered by the audio out interrupt service request. The audio renderer consumes a full buffer and produces an empty buffer.

Stop Components

```
tmolRendererStop(rendInst);
tmolProcessStop(procInst);
tmolDigitizerStop(digInst);
```

Since the components are running autonomously, either in a task or an ISR, it is up to the application programmer to decide what causes components to be stopped. This should be built into the application. The application calls **tmol*Com*Stop** of each component to stop its processing. This function acts synchronously. In other words, it will not return until the component has exited its processing loop. During a component's stop sequence, it will expel all internally held packets. Therefore, after all components are stopped, all packets should be in the empty queues. More detailed information on stopping can be found in Chapter 10, *TSSA Component Basics*.

Closing Instances

```
tmolRendererClose(rendInst);
tmolProcessClose(procInst);
tmolDigitizerClose(digInst);
```

The application can start and stop various components numerous times. However, when a component is no longer used, the application should close it by calling **tmol*Com*Close**. The close function frees up all of the resources previously allocated in open.

Destroying InOutDescriptors

```
tsaDefaultInOutDescriptorDestroy(iodesc1);
tsaDefaultInOutDescriptorDestroy(iodesc2);
```

When the InOutDescriptors are no longer used, they must be destroyed to free the memory allocated in the **tsaDefaultInOutDescriptorCreate**. This is done in the function **tsaDefaultInOutDescriptorDestroy**.

More Advanced Issues

Delaying Setting of Format

If it is appropriate, an **InOutDescriptor** can be created with a Null format. This can be done when an application does not yet know the format of the connection. However, the format must be specified before the connection is used, either by the application or by the sender component. To install a format in an existing **InOutDescriptor**, the application can call the function **tsaDefaultInstallFormat**. For details about the component installing the format, refer to Chapter 10, *TSSA Component Basics*.

Connecting a TSSA Component to a Non-TSSA Component

When connecting a TSSA component to a non-TSSA component, create a default capabilities of type **tsaDefaultCapabilities_t** with the appropriate formats supported by the non-TSSA component, or simply with a generic format. If the non-TSSA component is to be the sender in this connection, set the **senderCap** to the default capabilities struct. Conversely, if the non-TSSA component is to be the receiver in the connection, set the **receiverCap** to the default capabilities struct. Then call **tsaDefaultInOutDescriptorCreate** to create an **InOutDescriptor** connecting the two components.

Note that the non-TSSA component must be able to understand TSSA packets. Immediately before starting the non-TSSA component, explicitly set the **senderState** or **receiverState** in the **InOutDescriptor** struct to **tsaIODescFlagACTIVE**. In TSSA components, these values are automatically set by the default layer.

Creating Packets Without InOutDescriptorCreate

Sometimes the standard packet setup service provided by **InOutDescriptorCreate** may not be appropriate. One example is when the application needs to create packets of different properties for circulation in the same connection. Another is when the application needs to set fields in the packet headers that are not set by **InOutDescriptorCreate**. In this case, the application must call **InOutDescriptorCreate** with the **numberOfPackets** field in the **InOutDescriptorSetup** struct to be **0**. Afterwards, create the packets as desired and put them on the empty queue of the **InOutDescriptor**.

In Place Processing

In TSSA, the data memory in a connection is recycled between two components. However, sometimes it is desired that the data memory pass through a component. This is called processing data “in place.”

When a component processes data "in place," it does not copy the memory from the input packet to the output packet. Instead, it copies a pointer to the data from one packet to another.

The application must be aware that a component processes in place and must set up the **InOutDescriptors** accordingly. First it should create the component's input descriptor as usual, with the appropriate buffer sizes. However, when creating the **InOutDescriptor** for the output connection, specify **numberOfBuffers** as needed, but specify each **bufsize** to be zero. No buffer memory will be allocated, but the packets with the appropriate number of buffers will be created and placed in the empty queues. Refer to Chapter 10, *TSSA Component Basics,* for more details about in place processing.

Configuring Components

TSSA components support four possible methods of configuration.

- The **tmol***Com***InstanceSetup** function sets up the component to run. This function should be called before **tmolComStart**. When **tmol***Com***InstanceSetup** is called while the component is running, the behavior is component-dependent.
- The **tmol***Com***InstanceConfig** function is the way of changing the state of a component while it is running. It uses a functional interface or a queue interface. When it uses a queue interface, it calls the function, **tsaDefaultInstanceConfig**, to put command messages on the control queue and wait for acknowledgment from the component. The control queue interface serves two purposes. First, it makes it possible to synchronize the configurations across processor boundaries. Also, it serves to synchronize the access to the internal variables of a component. The component checks the control queue only at well defined points in the code.
- The **tmol***Com***InstanceConfig** function is implemented as a functional interface to configuring the component. A functional interface may be preferred when configuration is highly time critical.
- Each component may include other functions in its API to control its operation. **tmol***Com***InstanceConfig** is the standard way.

The **capabilities** flags in the component's default capabilities struct indicates whether or not the component supports queue-based instance configuration. When an application plans to call a component's **tmol***Com***InstanceConfig** function, the application must provide the queues for the component at instance setup. This is done by calling **tsaDefaultControlDescriptorCreate** to create a control descriptor. Accordingly, when the control descriptor is no longer used, **tsaDefaultControlDescriptorDestroy** must be called to free the memory.

Reconnecting TSSA Components

The TSSA default functions provide primitives to support the dynamic reconnection of components. This feature is required to construct large systems that are not shut down when you request a change of configuration. Reconnect is accomplished using the functions **tsaDefaultSenderReconnect** and **tsaDefaultReceiverReconnect**. These are described in the documentation of tsa.h.

The reconnect functions are applied to an **InOutDescriptor**. The component to be reconnected must be stopped. The **receiverReconnect** function does not require a format parameter. The format of the connection is determined by the sender, and the receiver will discover its new format at instance setup, or upon receipt of the first packet. However, the **senderReconnect** function does require a format parameter.

When reconnecting both ends of an **InOutDescriptor**, **receiverReconnect** should be called first. This is because **senderReconnect** will check the requested format against the capabilities of the receiver.

For example, the DTV audio system reconnects InOutDescriptors when you request a new configuration.

Debugging TSSA Applications

TSSA programs are TriMedia programs. Therefore, they can be debugged using the TriMedia debugger and DP functions.

TSSA modules are available in debuggable versions. We strongly recommend that you use these in the early phases of your development. Aside from debugging the program, the **tmAssert** mechanism might catch trivial errors often made in component setup during initial development. Using the debuggable versions has been observed to speed the bringup of new TriMedia programs.

Most TSSA components also use DP internally. Although the released versions of each library are compiled with no DPs defined, it can be useful, for example, to turn on DP in a component or in tsaDefaults.c.

Finally, for TSSA programs using pSOS, the pSOS monitor in the TriMedia debugger can be used. For more information, see Chapter 19, *Debugging TriMedia pSOS+™ Applications*, of Book 4, *Software Tools*, Part C.

Chapter 9

Applications Using a Non-Streaming Model

Introduction

This chapter deals with the properties of applications using TriMedia AL libraries in non-streaming or push mode.

While the OL library interfaces of different application libraries are almost identical, the interfaces of AL layer non-streaming libraries vary. Its structure depends on the functionality of the library. It can have a similar interface to the corresponding OL layer library if it implements processing on streaming data like the AC-3 or Pro Logic audio decoders. Both provide the standard interface functions **Open**, **Close**, **GetInstanceSetup**, **InstanceSetup**, **InstanceConfig** and some special functions of the **processData** category. However, an AL layer non-streaming mode library interface could also consist of less standard functions and a variety of processData functions like the 2-D graphics library. The 2-D library is a very special case of a TSA library because its functions are context independent. That means the result of an individual function call depends only on the input data and nothing else. It represents a more classical form of a function library. On the contrary, the kind of processing of the **processData** functions of other libraries depends on the state of the library instance.

The main difference between an AL application and an OL application is the way the data input/output management is implemented during the processing phase. The OL application just sets up connections and starts all components. After that, all data transfer occurs automatically in a hidden way. In contrast, the AL application has to implement the data input/output processing itself.

The counterparts to the streaming mode **start** function are the **processData** functions, which implement a limited time processing. They get input data, process it, generate a certain output, and terminate. On the other hand, the **start** function of the OL interface implements an unlimited time processing. It synchronizes itself with connected components through the availability of data packets at the respective inputs and outputs. The concepts of AL applications are explained using an example program written in pseudo code.

DATPUSH1.MOV

DATPUSH2.MOV

Figure 10 Animations showing range of behavior of non-streaming component. Animation 1 is an example of a simple synchronous library working on data that does not fit the streaming model. Animation 2 is an example of a synchronous library working on a stream of data.

Sample Application

This example implements a data filter application. The input data originates from a file and the output data is written to another file.

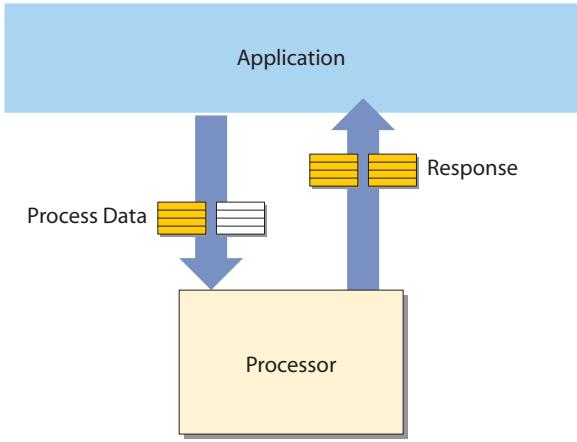


Figure 11 Passing a process data function to a non-streaming component

The pseudo code for such an application appears as follows:

```
/* Open component */
status = tmalFilterOpen( &FilterInstance);

/* Get instance setup structure for the component */
status = tmalFilterGetInstanceSetup( FilterInstance, &FilterSetup);

/* Change settings in the setup structure */
((ptmalAudioFormat)(FilterSetup->defaultInstSetup->
   inputDescriptor[0].format))->dataSubtype = apfStereo16;

/* Configure component with the modified setup struct */
status = tmalFilterInstanceSetup( FilterInstance, FilterSetup);

/* data processing loop */
while( !done ){
   status = fread( inputData, 1, INBUFSIZE, infile);
   if (status < INBUFSIZE) done = 1;       /* end of file reached */
/* set packet info for input packet */
   inputPacket.buffers[0].bufSize   = INBUFSIZE;
   inputPacket.buffers[0].dataSize  = status;
   inputPacket.buffers[0].data      = inputData;
/* set packet info for output packet */
   outputPacket.buffers[0].bufSize  = OUTBUFSIZE;
   outputPacket.buffers[0].dataSize = 0;
   outputPacket.buffers[0].data     = outputData;
   FilterData.input   = &inputPacket;
   FilterData.output  = &outputPacket;
   status = tmalFilterProcessData( FilterInstance, &FilterData);
   status = fwrite( outputData,1,outputPacket.buffers[0].dataSize,outfile );
```

```
}

/* Closing component */
tmalFilterClose( FilterInstance);
```

The following sections describe the individual parts of this code example in more detail. In the previous code example, such things as definition of variables and structs are left out to keep it as simple as possible. The following sections contain the definitions in the order of occurrence.

Open the AL Library Component

All services of AL library components are only available to an application through a unique instance handle. This instance handle is of the type **Int**. Instance handles are managed by the **tmal***Com***Open** and **tmal***Com***Close** functions of TSA components. The **tmalComOpen** function checks if another instance of the requested component is available. In this example, the integer **FilterInstance** gets the instance value assigned. If no more free instance is available the **tmal***Com***Open** function returns with the error message **TMLIBAPP_ERR_NO_FREE_INSTANCES**.

In most cases, the instance handle represents a pointer to the component's instance, variable which stores all information about the instance's state. Instances are a requirement for reentrant libraries unless the library does not store any states.

```
/* Variables and structs */
Int FilterInstance;

/* Open component */
status = tmalFilterOpen( &FilterInstance);
```

Getting Instance Setup Struct

The basic configuration of a TSA library component is done by the **tmal***Com***InstanceSetup** function. This function has a pointer to an instance setup struct as an argument. A convenient way to acquire such a struct is provided by the **tmal***Com***GetInstanceSetup** function. It fills in a pointer to a valid configuration. The application needs to change only those fields that don't match its requirements.

```
/* Variables and structs */
ptmalFilterSetup_t FilterSetup;

/* Get instance setup structure for the component */
status = tmalFilterGetInstanceSetup( FilterInstance, &FilterSetup);
```

The type **ptmFilterSetup_t** is a pointer to an instance variable of the component. The first field of all instance variables is always a pointer to the default instance setup.

Configuration of the Library Component

Most TSA library components need to be configured before the actual processing can be performed. The instance setup received from **tmalFilterGetInstanceSetup** contains the settings that are most likely to be required in the application. The necessary changes to the instance setup struct are applied before **tmalFilterInstanceSetup** is called. In the example code, the data subtype code of the input format is set to 16 bit stereo. Note that the format pointer of the input/output descriptors is a void pointer. A type cast is therefore required to access the members of the format struct.

```
/* Change settings in the setup structure */
((ptmalAudioFormat) (FilterSetup->defaultInstSetup->
   inputDescriptor[0].format))->dataSubtype = apfStereo16;
```

Set Up of the Library Component

After the instance setup struct is adapted to the application's specific needs the component gets configured by the call of **tmalFilterInstanceSetup**. The component is configured successfully if the return value is **TMLIBAPP_OK**. Otherwise an incorrect setup struct is used.

IMPORTANT
TSA libraries discover which input and output pins are active by checking whether or not the associated queues are null.

This means that even if no streaming is desired, values must be assigned to the respective empty and full queue fields of the input/output descriptors. In this example, it is assumed that the **tmalFilterGetInstanceSetup** function fills the respective queue fields with values.

```
/* Configure component with the modified setup struct */
status = tmalFilterInstanceSetup( FilterInstance, FilterSetup);
```

Data Processing

Now that the filter library is configured, the actual data processing can start. Up to now the steps undertaken have been very similar to those required within an OL library based application. In contrast to the OL layer example, no action has yet been taken toward preparing data packets that are to be sent to the component and received from the component. In an AL layer non-streaming mode application this normally happens in conjunction with the actual data processing. The more obvious difference is that the data processing is started by a simple function call (**tmol***Com***Start**) in the OL layer application and continues automatically. AL layer applications on the other hand must implement the handling of input and output data themselves.

The discussed example application reads a block of data from an input file, applies processing to it and generates a block of output data. This block of output data is then written to an output file. In this particular example, the process data function gets all

necessary information on the input and output data via its second parameter which is a struct containing pointers to the input packet and output packet. To keep this example simple, the packet header and packet format structs are omitted. However, they would normally not change during the processing phase.

The processing is stopped when the end of the input file is reached.

```
/* typedef from tmalFilter.h */
typedef struct tmalFilterIOStruct_t{
   ptmAvPacket_t input;
   ptmAvPacket_t output;
} tmalFilterIOStrcut_t, *ptmalFilterIOStruct;

/* variables and structs */
tmAvPacket_t       inputPacket, outputPacket;
tmalFilterIOStruct FilterData;
UChar              *inputData  = calloc( 1, INBUFSIZE);
UChar              *outputData = calloc( 1, OUTBUFSIZE);

/* data processing loop */
while( !done ){
   /* reading chunk of input data from file */
   status = fread( inputData, 1, INBUFSIZE, infile);
   if (status < INBUFSIZE) done = 1;     /* end of file reached */
   /* set packet info for input packet */
   inputPacket.buffers[0].bufSize   = INBUFSIZE;
   inputPacket.buffers[0].dataSize  = status;
   inputPacket.buffers[0].data      = inputData;
   /* set packet info for output packet */
   outputPacket.buffers[0].bufSize  = OUTBUFSIZE;
   outputPacket.buffers[0].dataSize = 0;
   outputPacket.buffers[0].data     = outputData;
   /* setting up argument struct for the process data function */
   FilterData.input   = &inputPacket;
   FilterData.output  = &outputPacket;
   /* process chunk of input data */
   status = tmalFilterProcessData( FilterInstance, &FilterData);
   /* writing chunk of output data to file */
   status = fwrite( outputData,1,outputPacket.buffers[0].dataSize,outfile );
}
```

All the processing implemented in the while loop corresponds to the functionality hidden in the start function of the streaming interface of a TSSA library. This example actually implements a sort of streaming processing by the means of using a non-streaming TSA library interface. The implementation of the while loop depends on the nature of the **processData** function(s). An application programmer must carefully study the documentation of the respective function. While some might implement internal buffering, others expect the input/output packets to have a certain granularity. There is no general rule on what processing is required around the **processData** function.

Closing the Component

When the instance of a TSA component is no longer used, the instance handle can be returned to the library by calling the component's **close** function. This function per-

forms a cleanup of all allocated memory resources. The freed instance is now available for a new user.

```
/* Closing component */
tmalFilterClose( FilterInstance);
```

Further Aspects of Using AL Libraries

More on Configuration

The entire configuration phase of the example application is very similar to the initialization of the OL library components described in Chapter 2, *Standard C Library*. However, this does not necessarily need to be the case for all AL layer non-streaming mode libraries. A setup of an AL component is only necessary when the library requires internal states as input/output configurations and special settings.

If a certain functionality can be implemented without any dependency on previous events, the functional interface of the library does not have to provide the standard functions **tmal***Com***Open**, **tmal***Com***GetInstanceSetup**, **tmal***Com***InstanceSetup**, **tmal***Com***InstanceConfig** and **tmal***Com***Close**. An example for such a library is the 2-D graphics library. The individual 2-D graphics library functions do not depend on any previous states. They implement a single buffer oriented processing, therefore, no context needs to be preserved.

Reconfiguration During Processing

While **tmal***Com***InstanceSetup** is used for the static configuration of the library component, **tmal***Com***InstanceConfig** is used to change certain settings of the configuration during the processing phase. The setup function is called only once before the real processing starts. It is responsible for the configuration of the input and output pins of the components. The component's internal behavior aside from input/output configuration can be modified by help of the **tmal***Com***InstanceConfig** function.

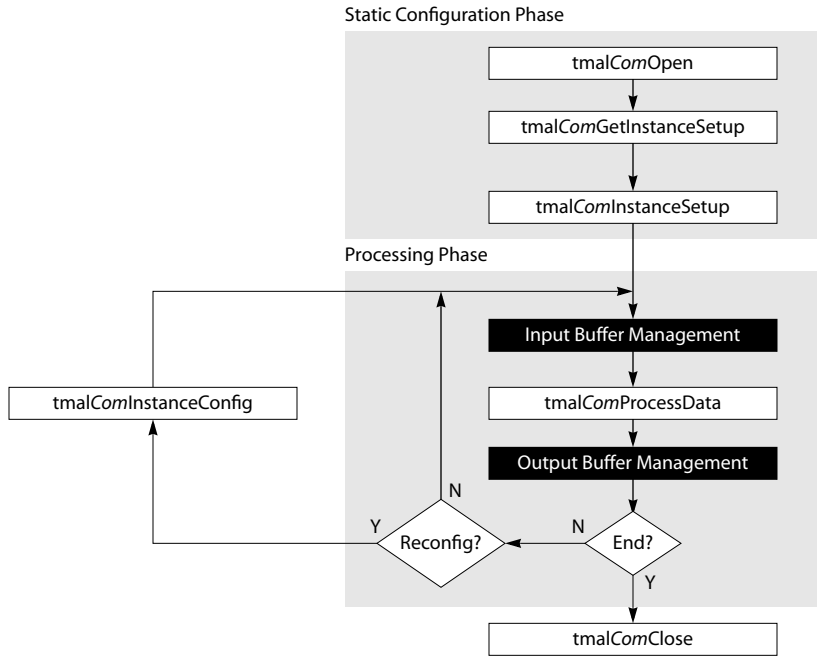


Figure 12 Non-streaming process flowchart

Figure 12 depicts the flow of a typical application applying some sort of processing to a continuous data stream using the non-streaming mode interface of a TSA library. White boxes represent calls to library functions and dark grey boxes stand for functionality that needs to be implemented within the application. Note that the processing loop implemented in the "Processing Phase" block corresponds to the functionality implemented in the start function of a streaming mode interface component. A difference is that the start function does not call the **tmal*Com*InstanceConfig** function directly. This must be implemented by the application programmer external to the start function.

Using AL Libraries in Streaming Mode

Some TSSA libraries support a streaming mode interface at the AL layer. It is possible to develop applications based on this interface even though it is not recommended. The application would have to provide implementations of the **datain** and **dataout** callback functions itself in that case.

The following two examples show possible implementations of both callback functions. Note that they must be dedicated to one instance of the TSSA library because the shown

functions are not re-entrant. Both functions access files that are assumed to be opened and closed somewhere else.

Sample Datain Callback Function

Such a **datain** callback function could look as follows:

```
/* Global variable used to indicate that allocated mem can be freed */
Int EOF_reached = 0;

tmLibappErr_t
datainFunction(Int instId, UInt32 flags, ptsaDatainArgs_t args){
   static Int              iniFlag;
   static tmAudioFormat_t  InFormat = {sizeof(tmAudioFormat_t), avdcAudio,
                                       atfLinearPCM, apfStereo16, 0, 0};
   static tmAvHeader_t     InHeader = {0, 0, NULL, NULL, NULL, {0,0},
                                       &InFormat};
   static tmAvPacket_t     InPacket = {&InHeader, 1, 0, {0,0,NULL}};
   static Address          inputData
   Int                     status;

   if( !iniFlag ){
      iniFlag = 1;
      inputData = (Address) _cache_malloc( INBUFSIZE, -1);
      InPacket.buffers[0].bufSize = INBUFSIZE;
   }
   if (flags & tsaDatainGetFull) {
      args->packet = &InPacket;
      status = fread(buffer, 1, INBUFSIZE, infile);
      if (status == 0) {
          EOF_reached = 1;
          free( inputData);
          tmalFilterStop( instId); /* stop filter */
      }
      InPacket.buffersInUse        = 1;
      InPacket.buffers[0].data     = inputData;
      InPacket.buffers[0].dataSize = status;
   }else { /* tsaDatainPutEmpty */
      /* free memory when end of stream is reached */
      if( EOF_reached) free( inputData);
   }
   return TMLIBAPP_OK;
}
```

This example does three different things:

- It allocates memory for the input data when it is first called.
- It checks if the caller requests a new data packet or returns a used one. In the case that the **start** function (in which the **datain** function calls occur) wants to acquire a new data packet, the **datain** callback function reads **INBUFSIZE** bytes from the input file and sets the data pointer of the packet to the read data.
- It declares the first packet buffer as used and sets the buffer's data size to the number of bytes actually read from the file.

If the end of file is reached the global **EOF_reached** is set to one to signal allocated data memory of the **datain** and **dataout** callback functions can be freed, see also the imple-

mentation of the **dataoutFunction** that follows. After that, the stop function of the TSA library component is called. This forces the component to leave the loop within its start function.

Whenever the start function returns a used packet to the **datain** callback function, no processing happens unless the end of the file is reached. It then frees the allocated buffer memory.

Sample Dataout Callback Function

The example implementation of the dataout callback function equals that of the previously described datain function. Upon first call, it allocates memory for the empty data packet. It then checks whether a full packet is sent or an empty packet is requested by the component's start function. If a full packet is sent, it is written to the output file. In the case where an empty packet is to be returned to the start function, the packet's first buffer data pointer is set to the beginning of the allocated memory, the data size is set to zero and the buffer is marked as unused. When the dataout function recognizes that no further packets will be acquired by the library, it frees the allocated buffer memory. By use of the global **EOF_reached**, it is possible to allocate and free memory only once.

```
/* Global variable used to indicate that allocated mem can be freed */
extern Int EOF_reached;

tmLibappErr_t
dataoutFunction(Int instId, UInt32 flags, ptsaDataoutArgs_t args){
   Int                     status;
   static Int              iniFlag = 0;
   static tmAudioFormat_t  outFormat = {sizeof(tmAudioFormat_t), avdcAudio,
                                        atfLinearout, apfFiveDotOne16, 0, 0};
   static tmAvHeader_t     outHeader = {0, 0, NULL, NULL, NULL, {0,0},
                                        &outFormat};
   static tmAvPacket_t     outPacket = {&outHeader, 1, 0, {0,0,NULL}};
   static Address          outputData;
   ptmAvBufferDescriptor_t buffer = ((ptmAvPacket_t)args->packet)->buffers;

   if( !iniFlag ){
      outputData = (Address) _cache_malloc( OUTBUFSIZE, -1);
      outPacket.buffers[0].bufSize = OUTBUFSIZE;
      iniFlag = 1;
   }
   if (flags & tsaDataoutPutFull) {
      status = fwrite( buffer->data, 1, buffer->dataSize, outfile);
      if (status != buffer->dataSize) exit( -1 );    /* file write error */
      if (EOF_reached) free( buffer->data);
   }else{ /* send empty packet */
      outPacket.buffersInUse         = 0;
      outPacket.buffers[0].dataSize  = 0;
      outPacket.buffers[0].data      = outputData;
      args->packet                   = &outPacket
   }
   return TMLIBAPP_OK;
}
```

This sort of implementation assumes that the library returns a used packet before requesting a new one. A more generalized implementation of the **datain** and **dataout** function would allocate and free memory dynamically:

- **datain** allocates memory when the flag equals **tsaDatainGetFull** and it frees memory when the flag equals **tsaDatainPutEmpty**.
- **dataout** allocates memory when the flag equals **tsaDataoutGetEmpty** and it frees memory when the flag equals **tsaDataoutPutFull**.

Chapter 10

TSSA Component Basics

Introduction

The construction of a TSSA-compliant component involves many choices. TSSA provides only a framework within which many different types of components can be constructed. TSSA compliance enables components designed by different developers to work together through a common interface. Components can achieve TSSA compliance by conforming to a few rules. The purpose of this chapter is to collect and explain some basic rules and the reasons behind aspects of the internal structure of a TSSA component.

The TSSA framework plays an important part in the design of a component. The most common type of TSSA components are streaming and task-based, with operating system independent data processing cores. In this chapter, these attributes are framed and illustrated through the design choices made in a simple component called CopyIO. CopyIO can be used as a template for other components that conform to a similar model.

Finally, the streaming, task-based model is not applicable to all functions. Near the end of this chapter, some guidelines are presented to help you choose the appropriate software model for a given functionality.

The previous three chapters (beginning with Chapter 7, *TSSA Essentials*) are strongly recommended as background reading.

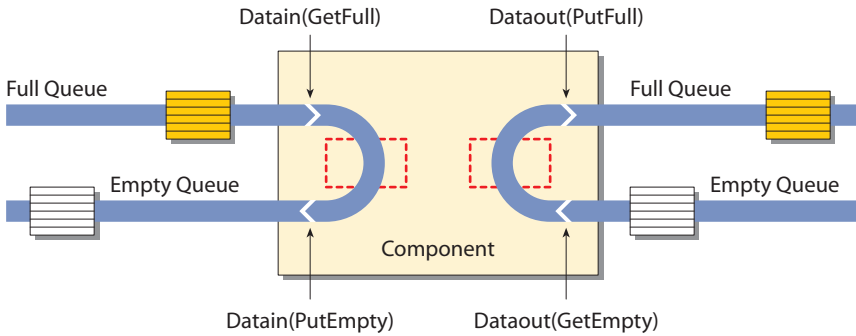


Figure 13 A Basic Streaming Component

Attributes of Common Components

TSA-compliant components can come in various forms. A component can be streaming or non-streaming. A streaming component acts on some data passing through it. A traditional library falls into the non-streaming model. A streaming component, however, can support both streaming and non-streaming modes. The data processing of a component can be operating system independent or dependent. A component can run in the context of a task, of an interrupt service routine, or of the application using the component. Task-based or ISR-based applies to streaming components, while the context of the appli-

cation pertains to a traditional library through a series of function calls. The most common TSA components are streaming, task-based, and have OS-independent data processing. CopyIO falls into this group of components.

The design process of a TSA component begins with the service that the component is to provide. Examples of the service are MPEG2 video decoding, audio rendering, or the parsing of HTML pages. Once the service is known, the component designer can begin applying the component to the various component attributes mentioned previously and that will be explained in the following section. CopyIO provides a service of copying data packets from input to output. With this in mind, we can apply CopyIO to a streaming, task-based, and OS-independent data processing model.

Streaming (Pull) and Non-Streaming (Push)

CopyIO is a streaming component because data passes through it. The data in this case are the packets to be copied. To support streaming, CopyIO must have an OL layer, which uses queues provided by an operating system to provide streaming. CopyIO is a streaming component which also supports a non-streaming mode. The streaming mode is also known as pull mode, and the non-streaming mode is also known as push mode.

The pull mode makes use of the concepts “start” and “stop.” In the pull mode, the application starts and stops the component. After being started by the application, the component “pulls” data from its input queue, and then processes the data. This loop of obtaining data, processing the data, and sending out the data happens in the function **tmalComStart**. Because the component needs to access a queue, it must have an OL layer. To use CopyIO in the pull mode, an application can call **tmolCopyIOStart** and **tmolCopyIOStop**.

In the push mode, instead of starting and stopping the component, the application asks the component to process some data. The application obtains the input data in some way, “pushes” the data into the component to be processed, and then sends out the processed data obtained by the component. By doing the task of obtaining and sending out data, the application implements the streaming mode, while using the component in the non-streaming mode. An application can use CopyIO in the push mode by calling **tmalCopyIOCopyPacket**.

OS-Independent Data Processing

The service provided by CopyIO is the copying of packets from input to output. Since the act of copying packets does not require the use of an operating system, the copying of the packets happens in the AL layer. In both streaming and non-streaming modes, the data process (i.e., the copying of packets) is done in the AL layer. In the streaming mode, the data processing is done in **tmalComStart**. In the non-streaming mode, the data processing is done in a function meant only to process data. In CopyIO, this function is called **tmalCopyIOCopyPacket**, whereas in other components, the function can have the

form **tmal***Com***ProcessData**. A component can also possess more than one **tmal***Com***-ProcessData** function.

Task-Based Context

Because CopyIO is a streaming component, it must be able to run in the context of its own task without waiting on the processor. By running in its own task during streaming operation, it is automatically put to sleep by the operating system, while another task is allowed to run when no data is available to be processed. Because tasks are provided by an operating system, the use of a task by a component is implemented in its OL layer. The application creates and starts the component's task when it calls **tmolComStart**, and stops and suspends the task when it calls **tmolComStop**. The task is finally destroyed in **tmolComClose**. Each task is associated with a function for the task to run in. The component's task is associated with the **tmalComStart** function.

TSSA Layers

OL Layer

The OL (Operating System) Layer is the part of a component that uses an operating system to achieve streaming and other OS-dependent functionality. The OL layer of a component is conceptually a thin shell around the AL layer component core. To create the OL layer of a component, you can copy the OL layer of a similar existing component and make minor changes.

Default Layer

The OL layer is connected to the AL layer by a default layer provided by TriMedia. This means that, for most TSSA API functions, the OL layer function calls the Default layer function, which calls the AL layer function. **GetInstanceSetup**, **InstanceConfig**, and **ProcessData** are exceptions to this. The default layer provides mechanisms common to the OL layer of all components and is, therefore, used only when the application uses the OL layer of a component. The purpose of the default layer is to remove from the component developer the responsibility of developing code that is common to all components, and is part of the TSSA framework.

For each instance of a component, the OL functions should be called by the application in the order shown in Figure 14.

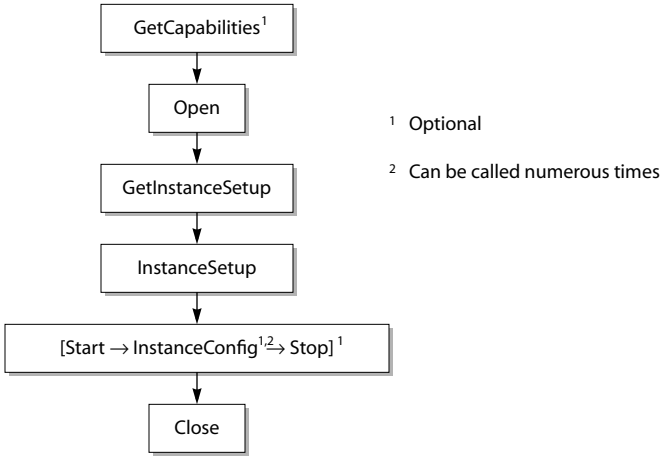


Figure 14 Order of OL Layer Function Calls

All default API functions operate on the **tsaDefaultInstVar** struct, which is created and initialized by **tsaDefaultOpen**. All default structures and function prototypes can be found in the header file tsa.h. The implementation of the default functions can be found in the file tsaDefaults.c.

AL Layer

The AL (Application) Layer is the processing core of a component. Unlike the OL layer, the AL layer is unique for each component and, therefore, cannot easily be copied from other components. AL functions are called either from corresponding default functions or directly from the application. When the application uses the OL layer, the AL functions are called from the default functions. Alternatively, the application can use the AL layer of a component by directly calling the AL functions. The AL layer API provides functions for streaming and non-streaming modes of the component. The streaming mode of a component is best accessed through its OL layer, while the non-streaming mode is only used in the AL layer.

For each instance of a component to be used for streaming operations, the AL functions should be called in the order shown in Figure 15.

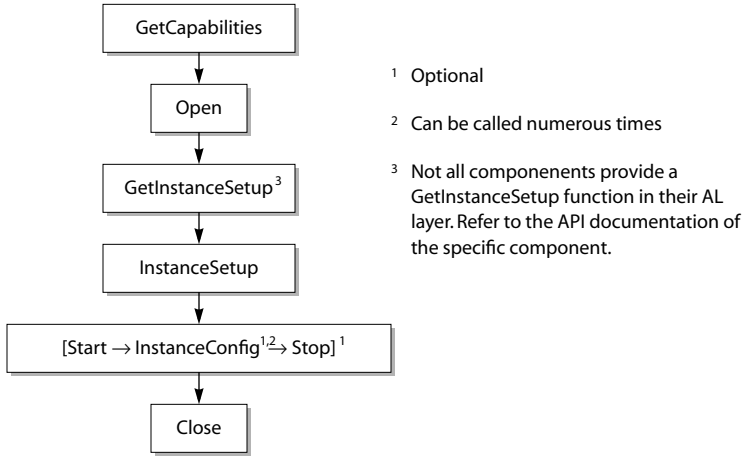


Figure 15 Order of AL Layer Function Calls (Streaming Operations)

For each instance of a component to be used for non-streaming operations, the AL functions should be called in the order shown in Figure 16.

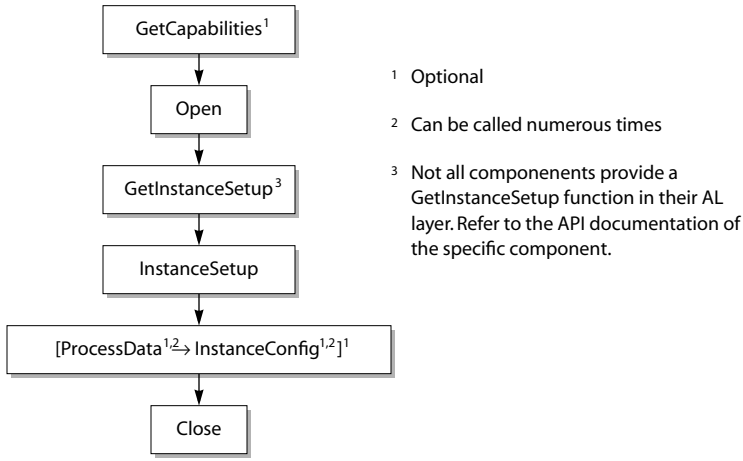


Figure 16 Order of AL Layer Function Calls (Non-Streaming Operations)

CopyIO Example and Explanation

The following section explains each of the TSSA API functions with the CopyIO component as an example. Each function is discussed in order of its OL, Default, and AL layers. Some key concepts of TSSA will be explained as they pertain to each API function of a component. These key concepts include instance setup parameters, instance configuration mechanisms, and component stop behavior.

GetCapabilities

OL GetCapabilities

```
tmLibappErr_t tmolCopyIOGetCapabilities(ptmolCopyIOCapabilities_t * cap){

   DP(("tmolCopyIOGetCapabilities()\n"));

   return tsaDefaultGetCapabilities(&add_done, &CopyIOTmalFunc,
      (UInt32 *) cap, TEXT_MEMORY_REQUIREMENT
      DATA_MEMORY_REQUIREMENT, PROCESSOR_REQUIREMENT);
}
```

The **tmol***Com***GetCapabilities** function is used by the application when using the OL layer to obtain information about the capabilities of a component before using it. A structure of the type **tmol***Com***Capabilities_t**, which contains the struct **tsaDefaultCapabilities** as its first field, is returned as a parameter. This structure is defined in tmolCom.h. For CopyIO, a structure of the type **tmolCopyIOCapabilities_t** is returned through a parameter. The following definition of the structure **tmolCopyIOCapabilities** is found in tmolCopyIO.h

```
typedef struct {
   ptsaDefaultCapabilities_t defaultCapabilities;
} tmolCopyIOCapabilities_t, *ptmolCopyIOCapabilties_t;
```

The **tmolCopyIOCapabilities** struct contains the **tsaDefaultCapabilities** struct as its first field. Note that this function should not be called from an ISR.

Default GetCapabilities

```
extern tmLibappErr_t tsaDefaultGetCapabilities (
   Bool               *add_done,
   ptsaDefaultFuncs_t  tsaFunc,
   UInt32             *cap,
   UInt32              text,
   UInt32              data,
   UInt32              proc
);
```

The **tsaDefaultGetCapabilities** function is called by **tmol***Com***GetCapabilities**. It takes as parameters an **add_done** value, a **tsaDefaultFuncs** structure containing AL layer API functions, a pointer to a capabilities structure to be returned, and the text, data, and proces-

sor requirements. In tmalCom.c, **add_done** is a static variable that determines if the OL memory requirements have been added to the AL requirements.

tsaDefaultGetCapabilities function adds the OL and AL requirements the first time it is called. It is initially set to false and is set to true after one call to **tmol*Com*GetCapabilities**. **tsaDefaultFuncs** is passed as a parameter to **tsaDefaultGetCapabilities** because the information is not yet stored internally because there is not yet an instance associated with this component when this function is called. With this information, **tsaDefaultGetCapabilities** calls **tmal*Com*GetCapabilities** and simply returns the **tmal*Com*GetCapabilities** structure as a parameter.

```
typedef struct tsaDefaultCapabilities {
   tmComponentClass_t            commponentClass;
   tmVersion_t                   version;
   UInt32                        capabilityFlags;
   Int                           textmemoryRequirement;
   Int                           datamemoryRequirement;
   Int                           processorRequirement;
   UInt                          numSupportedInstances;
   UInt                          numCurrentInstances;
   UInt                          numberOfInputs;
   ptmAvFormat_t                *inputFormats;
   UInt                          numberOfOutputs;
   ptmAvFormat_t                *outputFormats;
   tsaReceiverFormatSetupFunc_t  receiverFormatSetup;
} tsaDefaultCapabilities_t, *ptsaDefaultCapabilities_t;
```

The information returned in the default capabilities structure includes the component class, the version information, capability flags, memory and processor requirements, the number of instances supported, the number of current instances, the number of inputs and outputs, and the format for each, and a receiver format setup function.

AL GetCapabilities

```
tmLibappErr_t tmalCopyIOGetCapabilities(ptmalCopyIOCapabilities_t * cap){
   DP(("tmalCopyIOGetCapabilities()\n"));
   *cap = &lcap;
   return TMLIBAPP_OK;
}
```

The **tmal*Com*GetCapabilities** function simply returns, through a parameter, the statically allocated and initialized **tmal*Com*Capabilities** structure. Note that it must return TMLIBAPP_OK for the **tsaDefaultGetCapabilities** function to work. For CopyIO, a structure of type **tmalCopyIOCapabilities** is returned. The following definition of the **tmalCopyIOCapabilities** struct is found in tmalCopyIO.h.

```
typedef struct {
   ptsaDefaultCapabilities_t defaultCapabilities;
} tmalCopyIOCapabilities_t, *ptmalCopyIOCapabilties_t;
```


The static initialization of the previous structure (**lcap**) is found in tmalCopyIO.c. **tmalCopyIOCapabilities** contains **tsaDefaultCapabilities (def_cap)** as its first field.

```
static tsaDefaultCapabilities_t def_cap = {
   ccGenericIn,                        /* component class       */
   {1,1,0},                            /* version               */
   tsaCapFlagsSupportsControlQueue,    /* capabilityFlags       */
   TEXT_MEMORY_REQUIREMENT,
   DATA_MEMORY_REQUIREMENT,
   PROCESSOR_REQUIREMENT;
   NUM_SUPPORTED_INSTANCES,
   0,                                  /* numCurrentInstances   */
   COPYIO_NUMBER_OF_INPUTS,
   id_format,                          /* inputFormats          */
   COPYIO_NUMBER_OF_OUTPUTS,
   od_format,                          /* outputFormats         */
   Null                                /* receiverFormatSetup   */
};
static tmalCopyIOCapabilities_t lcap = {
   &def_cap
};
```

The **TEXT_MEMORY_REQUIREMENT**, **DATA_MEMORY_REQUIREMENT**, and **PROCESSOR_REQUIREMENT** of the OL and AL layers of the component are defined at the top of tmolCom.c and tmalCom.c, respectively. **COM_NUMBER_OF_INPUTS** and **COM_NUMBER_OF_OUTPUTS** are defined in tmalCom.c.

Open

OL Open

```
tmLibappErr_t tmolCopyIOOpen(Int * instance){
   pInstVars_t   ivp;
   tmLibappErr_t rval = TMLIBAPP_OK;

   DP(("tmolCopyIOOpen(): "));

/* memory for the instance variable */
   ivp = (pInstVars_t) calloc(1, sizeof (InstVars_t));
   if (!ivp) return TMLIBAPP_ERR_MEMALLOC_FAILED;

/* Create setup space for default initializations */
   ivp->setup =
   (ptmolCopyIOInstanceSetup_t)calloc(1,sizeof(tmolCopyIOInstanceSetup_t));
   if( !ivp->setup ){
      free(ivp);
      return TMLIBAPP_ERR_MEMALLOC_FAILED;
   }

/* allocates default instance variable, along with default instance setup */
   rval =
   tsaDefaultOpen(&(ivp->defInstVars),INST_OPEN_MAGIC,&CopyIOTmalFunc);
   if( rval ){
      free(ivp);
      return rval;
   }
   ivp->setup->defaultSetup = ivp->defInstVars->instSetup;
   ivp->setup->delay = 0;

   strcpy(ivp->defInstVars->instSetup->taskName, "COPY");
   ivp->defInstVars->instSetup->stackSize  = DEFAULT_STACK_SIZE;
   ivp->defInstVars->instSetup->taskFlags =
   tmosTaskFlagsPreempt | tmosTaskFlagsNoSliced | tmosTaskFlagsStandard;

   DP(("created instance %x\n", ivp));
   *instance = (Int) ivp;

   return rval;
}
```

The **tmol***Com***Open** function is used by the application when using the OL layer to allocate an instance of the component for usage. **tmol***Com***Open** first allocates memory for the **InstVars** struct of the OL layer. For CopyIO, the definition of the OL **InstVars** struct is found in tmolCopyIO.c.

```
typedef struct inst_vars {
   ptsaDefaultInstVar_t        defInstvars;
   ptmolCopyIOInstanceSetup_t  setup;
} InstVars_t, *pInstVars_t;
```

Note
TSA provides the flexibility for a component to have different sets of instance variables for the OL and AL layers if needed. Therefore, an InstVars structure is defined in each layer.

After allocating memory for its **InstVars** struct, **tmol***Com***Open** then allocates memory for the setup field in the **InstVars** struct. (Refer to *InstanceSetup* on page 71 for more information). Next, **tmol***Com***Open** sends the **defInstVars** field in the **InstVars** struct to **tsaDefaultOpen** to be allocated and initialized. After **ivp->defInstVars->instSetup** has been allocated, **ivp->setup->defaultSetup** is manually set to point to the same memory.

Subsequently, **tmol***Com***Open** sets the **taskName**, **stackSize**, and **taskFlags** in the **defInstVars** to the appropriate values for the component, to be used in **tsaDefaultInstanceSetup** to create the component task. Note that for ISR components, the field **createNoTask** should be set to true here. Finally, the pointer to the **InstVar** struct is returned as the instance id.

Note that this function should not be called from an ISR.

Default Open

```
extern tmLibappErr_t tsaDefaultOpen (
   ptsaDefaultInstVar_t  *divp,
   UInt32                magic,
   ptsaDefaultFuncs_t    tsaFunc
);
```

The default open function first allocates a **tsaDefaultInstVar** structure.

```
typedef struct tsaDefaultInstVar {
   UInt32                    magic;
   ptsaDefaultInstanceSetup_t instSetup;
   Int                       tmalInstance;
   ptsaDefaultFuncs_t        tmalFunc;
   volatile tsaTaskStatus_t  taskstatus;
   UInt32                    task;
   UInt32                    numberOfInputs;
   UInt32                    numberOfOutputs;
   UInt32                    stopSemaphore;
   UInt32                    configSemaphore;
   Int                       periodOfComponent;
   UInt32                    reserved;
} tsaDefaultInstVar_t, *ptsaDefaultInstVar_t;
```

The **tsaDefaultInstVar** structure contain information pertaining to the default functionality of the instance. **magic** is used by the component to check the validity of the instance. **instSetup** allows the application set the default functionality of the instance to the desired behaviors. **tsaDefaultOpen** allocates the default instance setup struct (**instSetup**). **tsaDefaultOpen** allocates it and sets its fields to default values.

The **inputDescriptors** and **outputDescriptors** fields of the default instance setup struct are arrays of pointers to **InOutDescriptors**. In **tsaDefaultOpen**, the array is allocated and the pointers initialized to Null. (Refer to *InstanceSetup* on page 71 for more information).

tmalInstance holds a pointer to the AL instance variable, to be allocated by **tmal***Com***Open**. **tmalFunc** contains all the AL functions needed by the defaults. It is stored in the

default layer in the **tsaDefaultInstVar** structure during **tsaDefaultOpen**. An example of the statically initialized structure **CopyIOTmalFunc**, is found in tmolCopyIO.c.

```
static tsaDefaultFuncs_t CopyIOTmalFunc = {
   (tsaGetCapabilitiesFunc_t) tmalCopyIOGetCapabilities,
   (tsaOpenFunc_t)            tmalCopyIOOpen,
   (tsaCloseFunc_t)           tmalCopyIOClose,
   (tsaInstanceSetupFunc_t)   tmalCopyIOInstanceSetup,
   (tsaStartFunc_t)           tmalCopyIOStart,
   (tsaStopFunc_t)            tmalCopyIOStop,
   (tsaInstanceConfigFunc_t)  tmalCopyIOInstanceConfig
};
```

The **taskstatus** is the internal value of the task status. **task** stores the task id returned from **tmosTaskCreate** in **tsaDefaultStart**, and is initially set to 0. **numberOfInputs** and **numberOfOutputs** are copied from the component's default capabilities structure.

stopSemaphore is used in the default stop sequence and **configSemaphore** is used with default implementation of the control queues. They are both created in **tsaDefaultOpen**. After initializing some values in **instSetup**, **tsaDefaultOpen** calls **tmalComOpen**.

Finally, **periodOfComponent** specifies how long a sender component waits before getting a packet from the empty queue, when the receiver component is stopped. By default, this value is 1, so that the sender component task will swap out and give other tasks a chance to run.

AL Open

```
tmLibappErr_t tmalCopyIOOpen(Int * instance){
   pInstVars_t ivp;

   DP(("tmalCopyIOOpen(): "));

/* memory for the instance variable */
   ivp = (pInstVars_t) calloc(1, sizeof(InstVars_t));
   if (!ivp) return TMLIBAPP_ERR_MEMALLOC_FAILED;

/* Create setup space for default initializations */
   ivp->setup =
   (ptmalCopyIOInstanceSetup_t)calloc(1,sizeof(tmalCopyIOInstanceSetup_t));
   if( !ivp->setup ){
      free(ivp);
      return TMLIBAPP_ERR_MEMALLOC_FAILED;
   }

   ivp->setup->defaultSetup =
   (ptsaDefaultInstanceSetup_t)calloc(1,sizeof(tsaDefaultInstanceSetup_t));
   if( !ivp->setup->defaultSetup ){
      free(ivp->setup);
      free(ivp);
      return TMLIBAPP_ERR_MEMALLOC_FAILED;
   }
   memset(ivp->setup->defaultSetup,'\0',sizeof(tsaDefaultInstanceSetup_t));

   ivp->setup->delay     = 0;
   ivp->setup->TimSleep  = Null;
   ivp->componentState   = tsaCompStateNotStarted;
   ivp->copyInProgress   = False;
   ivp->inPacket         = Null;
   ivp->outPacket        = Null;
   ivp->delay            = 0;
   ivp->TimSleep         = Null;

   *instance = (Int) ivp;
   def_cap.numCurrentInstances++;

   DP(("created instance %x\n", ivp));
   return TMLIBAPP_OK;
}
```

Similar to the OL layer, the AL layer has an **InstVars** struct that is used by an instance of the component. For CopyIO, its definition is found in tmalCopyIO.c.

```
typedef struct inst_vars {
/* common fields */
   UInt32               parent;
   tsaCompState_t       componentState;
/* callback functions */
   tsaDatainFunc_t      datainFunc;
   tsaDataoutFunc_t     dataoutFunc;
   tsaCompletionFunc_t  completionFunc;
   tsaControlFunc_t     controlFunc;
/* specific to CopyIO */
   Bool                 copyInProgress;
   ptmAvPacket_t        inPacket;
   ptmAvPacket_t        outPacket;
```

```
   UInt32              delay;
   tsaTimSleepFunc_t   TimSleep;
} InstVars_t, *pInstVars_t;
```

The memory for the AL **InstVars** struct is allocated and filled with default values. These values can be overwritten when **tmal***Com***InstanceSetup** is called. Streaming components must have a component state to know when to exit the processing loop in **tmal***Com***Start**. Callback functions used by the component, and the parent field used to identify the owner of the callback functions, should be copied from the **tsaDefaultInstanceSetup** struct in **tmal***Com***InstanceSetup**. Pointers to packets used by the component also belong in the **InstVars** struct, as each instance of the component should have its own set of packets.

Basic Reentrancy

There are some variables that are global to all instances of a component. To enable basic reentrancy of components, read and modify access to these variables must be protected.

One example of these global variables is the **numCurrentInstances** in the default capabilities flags. For components that support one or a limited number of instances concurrently, this variable must be protected in **tmalComOpen** and **tmalComClose** when it is read and modified. To protect the variable, it is recommended that task scheduling be turned off during and between the reading and modifying of this variable. On TriMedia, this is done with the functions **AppModel_suspend_scheduling** and **AppModel_resume_scheduling**. (For more information, see Chapter 1, *TriMedia Utility Functions*, of Book 5, *System Utilities*, Part A.)

AppModel.h must be included before using these functions. Note that all variables global to all instances of a component and read/modify sensitive must also be protected in a similar way to ensure basic reentrancy.

Task 1

```
#include <tmlib/AppModel.h>;
...
AppModel_suspend_scheduling();
lcap->numCurrentInstances++;
AppModel_resume_scheduling();
```

Task 2

```
#include <tmlib/AppModel.h>;
...
AppModel_suspend_scheduling();
lcap->numCurrentInstances++;
AppModel_resume_scheduling();
```


GetInstanceSetup

OL GetInstanceSetup

```
tmLibappErr_t tmolCopyIOGetInstanceSetup(
   Int instance, ptmolCopyIOInstanceSetup_t *setup
){
   pInstVars_t ivp = (pInstVars_t) instance;

   DP(("tmolCopyIOGetInstanceSetup(%x)\n", instance));
   tsaCheckOpen(ivp->defInstVars, INST_OPEN_MASK, INST_OPEN_MAGIC);

   *setup = ivp->setup;
   return TMLIBAPP_OK;
}
```

The **tmol***Com***GetInstanceSetup** function provides the application that is using the OL layer an instance setup structure that has previously been initialized by **tmol***Com***Open** with default values. By calling this function to obtain a template instance setup structure before setting up a component, the application does not have to set fields that should have default values. **tsaCheckOpen** checks if the component has been opened. A pointer to the setup field in the **InstVars** struct is returned through a parameter.

AL GetInstanceSetup

```
tmLibappErr_t tmalCopyIOGetInstanceSetup(
   Int instance, ptmalCopyIOInstanceSetup_t *setup
){
   pInstVars_t ivp = (pInstVars_t) instance;

   DP(("tmalCopyIOGetInstanceSetup(%x)\n", instance));

   *setup = ivp->setup;
   return TMLIBAPP_OK;
}
```

The **tmal***Com***GetInstanceSetup** function to provide an AL instance setup template for applications using the component in the AL layer. The function returns a structure of the type **tmal***Com***InstanceSetup_t**. The application can call this function to obtain a template instance setup structure when using the AL layer. Refer to the API reference of the specific component for more details.

InstanceSetup

The InstanceSetup function is used to set up a component before it is started, when it is not running. It must be called as least once before **tmol***Com***Start** is called. To configure a component while it is running, the application should call **InstanceConfig**. (See *InstanceConfig* on page 84 for more information).

OL InstanceSetup

```
tmLibappErr_t tmolCopyIOInstanceSetup(
   Int instance, tmolCopyIOInstanceSetup_t *setup
){
   pInstVars_t   ivp  = (pInstVars_t) instance;
   tmLibappErr_t rval = TMLIBAPP_OK;

   DP(("tmolCopyIOInstanceSetup(%x)\n", instance));
   tmAssert(setup, TMLIBAPP_ERR_INVALID_SETUP);
   tsaCheckOpen(ivp->defInstVars, INST_OPEN_MASK, INST_OPEN_MAGIC);

   if( !setup->TimSleep ) setup->TimSleep = tmosTimSleep;

   rval = tsaDefaultInstanceSetup(instance, (UInt32 *)setup);
   tmAssert(!rval, rval);                    /* catch in debug mode */
   if( rval ) return rval;                   /* catch in release mode too! */

   ivp->defInstVars->magic |= INST_SETUP_MAGIC;
   return rval;
}
```

tmolComInstanceSetup is called by the application when using the OL layer to set up an instance of the component. For CopyIO, the type definition of the setup field, **tmolCopyIOInstanceSetup**, is found in tmolCopyIO.h.

```
typedef struct {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   UInt32                      delay;
   tsaTimSleepFunc_t           TimSleep;
} tmolCopyIOInstanceSetup_t, *ptmolCopyInstanceSetup_t;
```

The **tsaDefaultInstSetup** struct is the first field of the **tmol***Com***InstanceSetup** struct, because TSA component-specific structures must have their corresponding default structures as the first field. For CopyIO, the **tmalCopyIOInstanceSetup** struct is exactly the same as the **tmolCopyIOInstanceSetup** struct. However, the **tmal***Com***InstanceSetup** struct and the **tmal***Com***InstanceSetup** struct can differ, depending on what information each layer of the component needs from the application.

setup, which is passed to **tsaDefaultInstanceSetup**, contains much information essential to the correct behavior of a component.

Note that this function should not be called from an ISR.

Default Instance Setup

```
extern tmLibappErr_t tsaDefaultInstanceSetup (
   Int       instance,
   UInt32   *setup
);
```

tsaDefaultInstanceSetup checks the validity of the instance and setup, then sets up the fields of the **tsaDefaultInstanceSetup** struct in the instance according to the values in

setup. Included in the **tsaDefaultInstanceSetup** struct are callback functions, input and output descriptors, and other variables used in the setting up of a component.

```
typedef struct tsaDefaultInstanceSetup {
   Int                        qualityLevel;
   tsaErrorFunc_t             errorFunc;
   UInt32                     progressReportFlags;
   tsaProgressFunc_t          progressFunc;
   tsaCompletionFunc_t        completionFunc;
   tsaDatainFunc_t            datainFunc;
   tsaDataoutFunc_t           dataoutFunc;
   tsaMemallocFunc_t          memallocFunc;
   tsaMemfreeFunc_t           memfreeFunc;
   ptsaClockHandle_t          clockHandle;
   ptsaInOutDescriptor_t     *inputDescriptors;
   ptsaInOutDescriptor_t     *outputDescriptors;
   UInt32                     parentId;
   tsaControlFunc_t           controlFunc;
   tsaControlDescriptor_t     controlDescriptor;
   UInt32                     priority;
   char                       taskName[16];
   UInt32                     stackSize;
   UInt32                     taskFlags;
   Bool                       createNoTask;
   UInt32                     taskStartArgument;
} tsaDefaultInstanceSetup_t, *ptsaDefaultInstanceSetup_t;
```

qualityLevel allows the application to set the quality level of the connection. This feature is currently not yet implemented in the default layer.

The TSSA callback functions allow the application to set certain behaviors of this instance of the component when the instance calls the callback functions. The callback functions include **errorFunc**, **progressFunc**, **completionFunc**, **DatainFunc**, **DataoutFunc**, **memallocFunc**, **memfreeFunc**, and **controlFunc**. These are discussed in *Callback Functions* on page 74.

The **inputDescriptors** and **outputDescriptors** fields of **tsaDefaultInstanceSetup** are arrays of pointers to **InOutDescriptor** structs. The memory for the array pointers are allocated in **tsaDefaultOpen** according to the number of inputs and outputs specified in the capabilities struct of the component, and then set to point to the correct **InOutDescriptor** structs created by **tsaDefaultInOutDescriptorCreate** by the application before calling **tmolComInstanceSetup**. The first input and output pins of the component are indexed 0 into the array. When a component has more than one input or output pin, the subsequent pins can be accessed by indexing into either **inputDescriptors** or **outputDescriptors**, respectively. See *InOutDescriptors* on page 74.

clockHandle allows the instance of the component to be associated with an instance of the **tsaClock**. (For more information, refer to Chapter 4, *Clock Support API*, of Book 5, *System Utilities*, Part A.) **priority**, **taskName**, **stackSize**, **taskFlags**, **createNoTask**, and **taskStartArgument** are used to configure the task to be created. Refer to *Start* on page 79 for more information.

Callback Functions

Because the AL layer of TSSA components is independent of the operating system, and because the function of communication between components is likely to be implemented using the facilities of an OS, the AL layer implements a set of callback functions. The standard set of callback functions is defined in the **tsaDefaultInstanceSetup** struct. Each of these functions uses the **parentId** to keep track of data structures used to communicate between the AL and the OL layers. The callback functions are defined in the higher level OL layer (or in the application) but they are called from the AL layer.
A set of default callback functions is provided as part of the TSSA default OL layer. The datain and dataout functions provided here allow all TSSA components to share the code and the mechanisms of communication.

Other standard callback functions include the error and progress functions, and the memory allocation functions. The error and progress functions would usually be defined by the application, as they provide a way for applications to insert code to be called in the context of the TSSA module. The memory management functions allow the application to control the mechanism used to allocate memory.

TSSA components usually use malloc to allocate a small instance variable during the call to the Open function. The amount of memory allocated here must be small, because it is always taken from the system pool. The next function, you call is the instance setup function, and here the memory allocation callback functions can be specified. If a TSSA component needs to allocate buffers, it must do so after you have specified these callback functions, and these callback functions must be used. In this way, you can control the source of the memory for each component. This allows application programmers to keep control over memory fragmentation. If no memory callback functions are specified, then the memory is taken from the system pool using _cache_malloc.

You might wonder why, at the OL layer, the component always calls the default callback functions rather than the one you installed. The default callback functions perform essential message handling, and they do this before dispatching your function, if it is installed. These messages are not passed on to your application. An example is tsaProgressFlagChangeFormat.

InOutDescriptors

In addition to callback functions, the **tsaDefaultInstanceSetup** struct contains descriptors for the input and output pins of the component. An **InOutDescriptor** structure describes the connection between two components. The most important fields of this structure are the full and empty queues, **senderState** and **receiverState**, and the format of the connection. The other fields further describe the connection and are used by the defaults. The following is the definition of **tsaInOutDescriptor_t**.

```
typedef struct InOutDescriptor {
   Pointer                 format;
   tsaInOutDescSetupFlags_t  flags;
   Bool                    receiverStopped;
   Bool                    cmdFullWakeupSent;
```



```
   Bool                      cmdEmptyWakeupSent;
   ptsaDefaultCapabilities_t senderCap;
   ptsaDefaultCapabiltiies_t receiverCap;
   UInt32                    senderIndex;
   UInt32                    receiverIndex;
   UInt32                    fullQueue;
   UInt32                    emptyQueue;
   ptmAvPacket_t             packetArray;
   ptmAvHeader_t             headerArray;
   UInt8                    *dataArray;
   UInt32                    packetBase;
   UInt32                    numberOfPackets;
   tsaIODescState_t          senderState;
   tsaIODescState_t          receiverState;
   ptmAvFormat_t             lastFormat;
   UInt32                    waitSemaphore;
   UInt32                    reserved;
} tsaInOutDescriptor_t, *ptsaInOutDescriptor_t;
```

By calling **tsaIODescriptorCreate** to create an instance of the **InOutDescriptor** struct and then passing it to the **InstanceSetup** of two components, an application effectively connects the two components. (The fields modified by **InOutDescriptorCreate** are **format**, **flags**, **senderCap**, **receiverCap**, **senderIndex**, **receiverIndex**, **fullQueue**, **emptyQueue**, **packetArray**, **headerArray**, **dataArray**, **packetBase**, and **numberOfPackets**. These are discussed in *InOutDescriptorCreate and InOutDescriptorDestroy* on page 75).

receiverStopped is checked by **Datain(GetFull)**. When **receiverStopped** is true, **Datain(GetFull)** will return **TMLIBAPP_NEW_FORMAT** after setting it to false. **cmdFullWakeupSent** and **cmdEmptyWakeupSent** prevent overflowing of queues with Wakeup packets. **lastFormat** is again checked against in **Datain(GetFull)** to determine whether or not to return **TMLIBAPP_NEW_FORMAT**. **senderState** and **receiverState** stores the state of the sender and receiver components. They determine the behavior of the Datain and Dataout functions, as well as the stop sequence. **waitSemaphore** is used by components that would wait on multiple queues one at a time. In this case, the component creates the semaphore and stores it in all the appropriate **InOutDescriptors** in **tmolComInstanceSetup**. It then waits on the semaphore instead of on a specific queue in **Datain(GetFull)**. The sender component releases the semaphore when it sends a packet to any one of the queues. **reserved** is not used and is for future extension.

InOutDescriptorCreate and InOutDescriptorDestroy

```
extern tmLibappErr_t tsaDefaultInOutDescriptorCreate (
   ptsaInOutDescriptor_t   *piodesc,
   pInOutDescriptorSetup_t  setup
);
```

At a minimum, the function **tsaIODescriptorCreate** allocates memory for a **InOutDescriptor**, creates the full and empty queues, and initializes the sender and receiver states. It takes as an argument a pointer to a **InOutDescriptorSetup** structure. It then creates an

InOutDescriptor structure according to the setup structure. The following is the definition of a **InOutDescriptorSetup**.

```
typedef struct InOutDescriptorSetup {
   ptmAvFormat_t              format;
   tsaInOutDescSetupFlags_t   flags;
   String                     fullQName;
   String                     emptyQName;
   UInt32                     queueFlags;
   ptsaDefaultCapabilities_t  senderCap;
   ptsaDefaultCapabilities_t  receiverCap;
   UInt32                     senderIndex;
   UInt32                     receiverIndex;
   UInt32                     packetBase;
   UInt32                     numberOfPackets;
   UInt32                     numberOfBuffers;
   UInt32                     bufSize[1];
} InOutDescriptorSetup_t, *pInOutDescriptorSetup_t;
```

If the application wants the connection to have a certain format of data, it will set the format field in the **InOutDescriptorSetup** struct to the desired format. If the format is provided by the application, **tsaDefaultInOutDescriptorCreate** checks if the two components are compatible using **senderCap**, **receiverCap**, **senderIndex**, and **receiverIndex**. **senderCap** and **receiverCap** are the capabilities structures of the sender and receiver in the connection.

senderIndex is the index into the **outputFormats** array in **senderCap**, and **receiverIndex** is the index into the **inputFormats** array in **receiverCap**. **tsaDefaultInOutDescriptorCreate** calls **tsaDefaultInstallFormat** to install the format in the **InOutDescriptor**.

The **flags** field indicates one or more properties of the **InOutDescriptor**. When used in the **InOutDescriptorSetup** structure, it can be **tsaIODescSetupFlagsCacheMalloc** and/or **tsaIODescSetupFlagsMultiProc**. **tsaIODescSetupFlagsCacheMalloc** causes **tsaDefaultInOutDescriptorCreate** to use **_cache_malloc** to create all memory. **tsaIODescSetupFlagsMultiProc** causes the **InOutDescriptor** to be created for use in a multiple processor environment, possibly between components on different processors. The flags from the setup are then copied to the **InOutDescriptor**.

Once the flags are copied from the setup struct, two flags can be added to it if they are found in the component capabilities structs. These flags are **tsaCapFlagsCopybackDatain** and **tsaCapFlagsInvalidateDataout**. These translate to **tsaIODescSetupFlagCopybackDatain** and **tsaIODescSetupFlagInvalidateDataout**, respectively, in the **InOutDescriptor**. The initial values of these flags are set for each **InOutDescriptor** when it is created by **tsaDefaultInOutDescriptorCreate**. The component is then free to modify the values as needed for each queue. This is done in the **InstanceSetup**.

Cache Coherency

The default functions **tsaDefaultInOutDescriptorCreate**, tsaDefaultDatain, and tsaDefaultDataout can act upon these flags to help components maintain cache coherency. If the sender component has an invalidate flag, **InOutDescriptorCreate** will copyback the data

buffers after creating them. Then, if the receiver component does not have a copyback flag, Dataout will invalidate the data buffers. If the sender does not have a invalidate flag and the receiver has a copyback flag, then Datain will copyback the data buffers. In all other cases, no invalidate or copyback operation is performed. Refer to the following table for clarification.

	Sender Invalidate	**Sender None**
Receiver Copyback	Copyback on create.	Copyback in Datain.
Receiver None	Copyback on create. Invalidate in Dataout if cache aligned.	Do nothing.

The application can specify the queue names and queue flags, but "**FULL**" and "**EMPTY**" will be used as queue names, and the default queue flags will be applied.

The **senderState** and **receiverState** are initialized to **ACTIVE** to avoid losing packets when the components are started. When the sender is started and the **receiverState** is **STOPPED**, the sender would put full packets directly into the empty queue, and therefore full packets containing data would be lost.

InOutDescriptorCreate will create the **numberOfPackets** number of packets, number them incrementally starting from **packetBase**, and put them onto the empty queue. Each packet is created with **numberOfBuffers** buffers. Each buffer has a size according to its index into the **bufSize** array. **bufSize**, by default, allows for one buffer per packet. See Chapter 8, "*Developing Applications Using a Streaming Model*," for more details on packet creation.

When **numberOfPackets** is 0, **tsaDefaultInOutDescriptorCreate** will create no packets and the application is expected to create the packets accordingly. Furthermore, if **numberOfBuffers** is 0, while **numberOfPackets** is not 0, the packets will be created, and the buffers are expected to be created later. **packetArray**, **headerArray**, and **dataArray** are used for optimization of packet memory allocation.

Formats

The format of data used by each pin is described in the format field of the **InOutDescriptor** struct. This format field can point to the **tmAvFormat** struct or any of its subclasses. Currently existing subclasses of **tmAvFormat** are **tmVideoFormat** and **tmAudioFormat** structs. **tmVideoFormat** and **tmAudioFormat** structs contain in addition to the fields of **tmAvFormat**, information pertaining especially to video or audio.

The definitions of **tmAvFormat** and its subclasses can be found in tm1/tmAvFormats.h in the include directory of the TriMedia Compilation System. This file is kept in TCS because it is used by the devices libraries, which are also found in TCS.

```
typedef struct tmAvFormat_t {
   UInt16            size;
   UInt16            hash;
   UInt32            referenceCount;
   tmAvDataClass_t   dataClass;
```

```
   UInt32            dataType;
   UInt32            dataSubtype;
   UInt32            description;
} tmAvFormat_t, *ptmAvFormat_t;
```

In addition to its usage in the **InOutDescriptor** struct, the format is also used in TSA packet headers (**tmAvHeader**) to describe the format of the data in the packet. Because of its usage in the packet header, it must conform to two rules that are required of memory pointed to from **tmAvHeader**.

First, it must be a structure, with **size** as its first field. It should be set to the size of the format structure used. For example, when the format of the data is described by **tmAvFormat**, the **size** field is set to **sizeof(tmAvFormat_t)**.

Second, there must not be any more levels of indirection in these structures (i.e., it must not contain any pointers). Currently existing data classes are **system**, **video**, **audio**, **control**, **generic**, and **other.** These can be found in the **tmAvDataClass** struct in tmAvFormats.h.

AL InstanceSetup

```
tmLibappErr_t
tmalCopyIOInstanceSetup( Int instance, tmalCopyIOInstanceSetup_t *setup ){
   pInstVars_t ivp = (pInstVars_t) instance;

   DP(("tmalCopyIOInstanceSetup(i:%x (p:%x))\n", ivp, ivp->parent));

/* common fields */
   ivp->parent         = setup->defaultSetup->parentId;

/* callback functions */
   ivp->datainFunc     = setup->defaultSetup->datainFunc;
   ivp->dataoutFunc    = setup->defaultSetup->dataoutFunc;
   ivp->completionFunc = setup->defaultSetup->completionFunc;
   ivp->controlFunc    = setup->defaultSetup->controlFunc;

/* specific fields */
   ivp->delay          = setup->delay;
   if(setup->TimSleep)
   ivp->TimSleep     = setup->TimSleep;

   return TMLIBAPP_OK;
}
```

The **tmal***Com***InstanceSetup** functions uses the information passed in from the **tmal***Com***InstanceSetup** struct to fill in the correct values for its **InstVars**. Like the **tmol***Com***InstanceSetup** struct, **tmal***Com***InstanceSetup** struct has **tsaDefaultInstanceSetup** struct as its first field.

Using OS Functions in the AL Layer

Sometimes a component needs to use some OS functions in its processing. For example, a component may need to use semaphores to lock data access or to receive hardware signals, or to use a timed sleep function, as in CopyIO. There are two ways to use OS func-

tions in the component's processing, depending on whether streaming should be supported in the AL layer and on how many OS functions are needed.

One is to send the OS functions to the AL layer as callback functions. This is done in CopyIO with **TimSleep**. The other is to move the entire processing loop to the OL layer, in **tmolComStart**. This is done in **VtransICP** with the use of a semaphore while waiting for the ICP device to be done. Streaming in the AL layer is not supported in **VtransICP**.

CopyIO uses **TimSleep** in **tmalCopyIOStart** to swap out for a set amount of time. It can be set when the application uses either the OL or AL layer. When the application uses the OL layer, it is given a chance to specify **TimSleep**, by setting **TimSleep** in the **tmolCopyIOInstanceSetup** struct to the desired function. If the application does not want to specify a **TimSleep** function, it is then set to **tmosTimSleep** in **tmolCopyIOInstanceSetup**. **tmalCopyIOInstanceSetup** then saves it as an instance variable. When the application uses the AL layer, it can similarly specify **TimSleep**, by setting **TimSleep** in the **tmalCopyIOInstanceSetup** to the desired function. If the application does not set it, then this function is not called in **tmalCopyIOStart**. Note that the AL function must check if your callback function has been set before calling it.

The other way to use OS functions in the component's processing is to move the entire processing to the OL layer. This is not recommended unless absolutely necessary (for example, if the processing needs many OS functions or if the component does not support streaming in the AL layer). The video transformer component **VtransICP** needs to use a semaphore to wait for the ICP device to finish processing. It does not support streaming in the AL layer and, therefore, does the main processing in the OL layer.

Start

OL Start

```
tmLibappErr_t tmolCopyIOStart(Int instance){
   pInstVars_t   ivp  = (pInstVars_t) instance;
   tmLibappErr_t rval = TMLIBAPP_OK;

   DP(("tmolCopyIOStart(i:%x (t:%x))\n", ivp, ivp->defInstVars->task));

   tsaCheckOpen(ivp->defInstVars, INST_OPEN_MASK, INST_OPEN_MAGIC);
   tsaCheckSetup(ivp->defInstVars, INST_SETUP_MASK, INST_SETUP_MAGIC);

   rval = tsaDefaultStart(ivp->defInstVars);
   return rval;
}
```

The **tmol***Com***Start** function is called by the application when using the OL layer to start the data processing of a component. It simply calls **tsaDefaultStart**, which will then complete the rest of the start sequence. It is essential to call **tsaDefaultStart** from any **tmol***Com***Start** function, as it ensures the correct starting of the component, by completing the start sequence.

Note that this function should not be called from an ISR.

Default Start

```
extern tmLibappErr_t tsaDefaultStart(ptsaDefaultInstVar_t divp);
```

tsaDefaultStart takes the **tsaDefaultInstVars** struct as argument. It first sets the **receiverState** of all its input descriptors and **senderState** of all its output descriptors to active. If **createNoTask** is set, **tsaDefaultStart** directly calls **tmal***Com***Start**. Otherwise, it creates and starts a task with **tsa_default_task**, if it does not already exist. **taskStartArgument** is given as an argument to the **tsa_default_task** function. By default, it is set to **tmalInstance**, but it can overwritten by the component in **tmol***Com***Start** before calling **tsaDefaultStart**.

The task created will have the properties specified in the default instance setup. These properties include the task name, stack size, and priority. If the task exists, it will be resumed instead of a new task being created. This avoids the overhead of creating and destroying a task for each start/stop sequence. The pSOS operating system runs the task with the highest priority. If the task has higher priority than the task starting it, it will run immediately. But if the task has a lower priority than the task starting it, it will wait until it is able to run. Keeping this in mind helps the application programmer prevent priority inversions.

Default Task

tsa_default_task is an internal function of the default implementation. When it is started as a task by **tsaDefaultStart**, it sets the **taskstatus** in the **tsaDefaultInstVars** to running. Then it calls **tmal***Com***Start** in which is the main processing loop. (See *AL Start* on page 81.)

tmal*Com***Start** does not exit until the application calls **tmol***Com***Stop** or it exits the loop with an error. When it exits the loop with an error, the system is inconsistent until the application actually calls **tmol***Com***Stop**. The rest of **tsa_default_task** is part of the stop sequence and is described in Default Stop on page 87.

AL Start

```
tmLibappErr_t tmalCopyIOStart( Int instance ){
   pInstVars_t          ivp       = (pInstVars_t)instance;
   tsaDatainArgs_t      di_args   = { COPYIO_MAIN_INPUT,  Null, 0 };
   tsaDataoutArgs_t     do_args   = { COPYIO_MAIN_OUTPUT, Null, 0 };
   tsaControlArgs_t     cargs     = { 0, 0, 0 };
   tsaCompletionArgs_t  comp_args = { 0 };
   UInt32               cmd;
   tmLibappErr_t        err       = TMLIBAPP_OK;

   DP(("tmalCopyIOStart(i:%x (p:%x))\n", ivp, ivp->parent));

   ivp->componentState = tsaCompStateRunning;

   while( ivp->componentState == tsaCompStateRunning ){
      DP(("CopyIO %x: Waiting for full packet\n", ivp));
      err = ivp->datainFunc(ivp->parent, tsaDatainGetFull |
                            tsaDatainWait | tsaDatainCheckControl, &di_args);
      ivp->inPacket = (ptmAvPacket_t)di_args.packet;
   /* gets TMLIBAPP_NEW_FORMAT on first packet */
      if( err==TMLIBAPP_NEW_FORMAT ) err = TMLIBAPP_OK;
      else if( err ) break;
      if( !di_args.packet ) continue;
      DP(("CopyIO %x: Received full packet %x\n", ivp,
         ivp->inPacket->header->id));
      ivp->inPacket->header->flags &= ~avhValidTimestamp;
   /* no valid timestamp */
      DP(("CopyIO %x: Waiting for empty packet\n", ivp));
      err = ivp->dataoutFunc(ivp->parent, tsaDataoutGetEmpty |
                          tsaDataoutWait | tsaDataoutCheckControl, &do_args);
      ivp->outPacket = (ptmAvPacket_t)do_args.packet;
      if( err ) break;
      if( !do_args.packet ) continue;
      DP(("CopyIO %x: Received empty packet %x\n", ivp,
         ivp->outPacket->header->id));
      ivp->outPacket->header->flags &= ~avhValidTimestamp;
   /* no valid timestamp */
      err = tmalCopyIOCopyPacket(instance, ivp->inPacket,
      ivp->outPacket);
      if( err ) break; /* break out of while (componentState) */

   /* wait delay ticks before sending back */
      if( ivp->TimSleep ){
         err = ivp->TimSleep(ivp->delay);
         if( err ) break; /* break out of while (componentState) */
      }

      DP(("CopyIO %x: About to send back in packet %x\n", ivp,
      ivp->inPacket->header->id));
      di_args.packet = ivp->inPacket;
      ivp->inPacket->buffersInUse = 0;
      ivp->datainFunc( ivp->parent, tsaDatainPutEmpty |
                       tsaDatainCheckControl, &di_args );
      ivp->inPacket = Null;

      DP(("CopyIO %x: About to send out packet %x\n", ivp,
         ivp->outPacket->header->id));
      cmd = tsaCmdDataPacket;
      do_args.packet = ivp->outPacket;
      ivp->dataoutFunc(ivp->parent, tsaDataoutPutFull |
```

```
                tsaDataoutCheckControl | tsaDataoutScheduleOnStop, &do_args);
      ivp->outPacket = Null;
   } /* end while componentState running */

   DP(("CopyIO %x: Exited start loop with command %x, err %x\n", ivp,
      cmd, err) );

/* stop sequence */
   if( ivp->inPacket ){
   /* put unprocessed packet back on datain empty queue */
      DP(("CopyIO %x: Expelling in packet %x into datain empty queue\n",
         ivp, ivp->inPacket->header->id));
      di_args.packet = ivp->inPacket;
      ivp->inPacket->buffersInUse = 0;
      ivp->datainFunc( ivp->parent, tsaDatainPutEmpty |
                       tsaDatainCheckControl, &di_args );
   }

   if( ivp->outPacket ){     /* put copied packets on full queue as empty */
      DP(("CopyIO %x: Expelling out packet %x into dataout full queue as \
         empty\n", ivp, ivp->outPacket->header->id));
      do_args.packet = ivp->outPacket;
      ivp->outPacket->buffersInUse = 0;
      ivp->dataoutFunc( ivp->parent, tsaDataoutPutFull |
                        tsaDataoutCheckControl, &do_args );
   }

/* signal completion */
   ivp->componentState = tsaCompStateStopCompleted;
   if( ivp->completionFunc ){
      comp_args.completionCode = err;
      err = ivp->completionFunc(ivp->parent, 0, &comp_args);
   }
   DP(("CopyIO %x: Completed\n",ivp));
   return err;
}
```

For a streaming component, **tmal***Com***Start** is the main processing loop of the component. It must loop on a **componentState**, which is set to **tsaCompStateRunning** before entering the loop. It will be eventually set to **tsaCompStateStopRequested** by **tmal***Com***Stop** when the component is stopped. Its function is to retrieve packets from the input full queue, process the data, and send the processed packets to the output full queue. For each IN pin of a component, **Datain(GetFull)** is called to obtain full packets from the full queue to process and **Datain(PutEmpty)** is called to put empty packets on the empty queue. Note that **Datain(PutEmpty)** must be called after a **Datain(GetFull)** to correctly "recycle" packets. Similarly, for each OUT pin of a component, **Dataout(GetEmpty)** is called to obtain an empty packet from the empty queue to fill and **Dataout(PutFull)** is called to put full packets on the full queue. Note that **Dataout(GetEmpty)** must be called before a **Dataout(PutFull)** to correctly "recycle" packets.

In each traversal of its processing loop, CopyIO copies packets from its one input pin to its one output pin. To do that, it first calls **Datain(GetFull)** to get a full packet to copy from. Then it calls **Dataout(GetEmpty)** to get an empty packet to copy to. Then it calls **tmalCopyIOPacket** to copy the packets. Other streaming components can replace this function with their own process data function. After copying the packet, it returns the

first packet back to the empty queue with a call to **Datain(PutEmpty)** and sends out the filled packet to the full queue will a call to **Dataout(PutFull)**.

Datain and Dataout

The flags given to Datain/out are essential to the correct behavior of a streaming component. For **Datain**, **tsaDatainGetFull** or **tsaDatainPutEmpty** indicates getting a packet from the full queue or putting a packet on the empty queue. Similarly for **Dataout**, **tsaDataoutPutFull** or **tsaDatainGetEmpty** indicates putting a packet on the full queue or getting a packet from the empty queue.

tsaDatainWait and **tsaDataoutWait** accompany **GetFull** or **GetEmpty** calls to indicate waiting on the queue until a packet arrives, on the input full queue or output empty queue, respectively. A timeout for this waiting can be indicated in the timeout field of the **tsaDatainArgs** or **tsaDataoutArgs**. **tsaDatainCheckControl** and **tsaDataoutCheckControl** indicates checking the control queue, after receiving a packet in a **GetFull** or **GetEmpty** call, if the component supports control queues.

tsaDataoutScheduleOnStop defines the behavior when the receiver of this **Dataout(PutFull)** is stopped. When the receiver of a **Dataout(PutFull)** call is stopped, the packets are put on the output empty queue instead of the full queue. If the component task calling **Dataout(PutFull)** has higher priority than the receiving task, it will run forever by getting packets from the empty queue and putting them back on the empty queue. To prevent this, **Dataout(PutFull)** has an option to schedule itself out of a specific time, and get other tasks a chance to run and the receiver component a chance to get out of the stopped state. With the **tsaDataoutScheduleOnStop**, the task is scheduled out after putting the full packet on the empty queue, according to the **periodOfComponent** field in the default instance variables. This field can be specified by the application. If it is not, a value of 1 is used.

After each Datain/out call, **tmalComStart** checks for error. The error returned can be real queue errors or informative errors, such as **TMLIBAPP_NEW_FORMAT** and **TMLIBAPP_STOP_REQUESTED**. **Datain(GetFull)** returns **TMLIBAPP_NEW_FORMAT** when a packet with a new format has arrived. This includes the first packet received from the input full queue. This is not a real error and therefore, the component should continue.

Exiting the Processing Loop

If there were any real errors or if the component has been requested to stop in the Datain/out function, the component must exit its processing loop. When exiting in a task-based component, the **tmal***Com***Start** function must expel all internally held packets. Packets from **Datain(GetFull)** should be put on the input empty queue as empty and packets from **Dataout(GetEmpty)** should be put on the output full queue as empty.

Because the expelling of the packets is done outside the processing loop, the instance variable that holds the packets must be set accordingly before checking the error from **Datain** or **Dataout**. For example, **inpacket** must be set to **di_args.packet** before checking

the error from **Datain(GetFull)**. Similarly, **outpacket** must be set to **do_args.packet** before checking the error from **Dataout(GetEmpty)**. This must be done to prevent losing packets when exiting the loop, because **inpacket** and **outpacket** are checked to determine if a packet needs to be expelled. Likewise, **inpacket** must be set to Null before checking the error from **Datain(PutEmpty)**, and outpacket must be set to Null before checking the error from **Dataout(PutFull)**, to prevent putting the packet back on the queues more than once.

After expelling any internally held packets, the component can call the completion function if desired. Note that at any point in the loop, the component can call the progress or error function to report to the application any event via the **progressCode** or **errorCode**, respectively.

InstanceConfig

InstanceConfig is called when the application wants to configure a component while it is running. It takes an argument of the type **tsaControlArgs_t** and its purpose is to change the instance variables of the AL layer to change the behavior of the running component. TSA provides the component developer the option to use a functional interface or queue interface for **InstanceConfig**, depending on how time-critical the configuration is and on the synchronization with the **tmal***Com***Start** function. If a configuration command is very time-critical, it is recommended to use the functional interface, as the change in configuration is immediate. However, if the **tmal***Com***Start** cannot handle having its instance variables changed without its knowledge (therefore requiring synchronization with the application), then the queue interface is recommended. With the queue interface, the **tmal***Com***Start** function will receive the configuration command when ready.

OL InstanceConfig

```
tmLibappErr_t
tmolCopyIOInstanceConfig(Int instance, UInt32 flags, ptsaControlArgs_t args){
   pInstVars_t   ivp  = (pInstVars_t) instance;
   tmLibappErr_t rval = TMLIBAPP_OK;

   DP(("tmolCopyIOInstanceConfig(i:%x (t:%x))\n",ivp,
      ivp->defInstVars->task));

   tmAssert(ivp, TMLIBAPP_ERR_INVALID_INSTANCE);
   tsaCheckOpen(ivp->defInstVars, INST_OPEN_MASK, INST_OPEN_MAGIC);
   tsaCheckSetup(ivp->defInstVars, INST_SETUP_MASK, INST_SETUP_MAGIC);

   rval = tsaDefaultInstanceConfig(ivp->defInstVars, flags, args);
   return rval;
}
```

Functional Interface

Using the functional interface, the **tmol***Com***InstanceConfig** calls the **tmal***Com***InstanceConfig** function directly to change the configuration of a component in the AL layer.

Queue Interface

Since the functional interface for the control queue is straightforward, CopyIO has an example of the queue interface, which is described below.

Using the queue interface, the **tmol*Com*InstanceConfig** assembles a control command packet and puts it on the component's command queue. It then waits for an acknowledgment from the response queue before exiting. To access the control and response queue, the **tmol*Com*InstanceConfig** function calls the default function **tsaDefaultInstanceConfig**.

Default InstanceConfig

```
extern tmLibappErr_t tsaDefaultInstanceConfig(
   ptsaDefaultInstVar_t  divp,
   UInt32                flags,
   ptsaControlArgs_t     args
);
```

The **tsaDefaultInstanceConfig** puts a command in the control queue and waits for acknowledgment from the response queue. A mutual exclusion semaphore prevents more than one task from trying to configure a component at the same time. In attempting to access the control queue, the task will block if it must wait for the component's semaphore. A timeout can be requested by setting the **tsaControlWait** flag. If the function times out, it returns with **TMLIBAPP_ERR_SEMAPHORE_TIMEOUT**. You can specify the timeout period, in ticks, in the **tsaControlArg** struct.

Because the component could be blocked waiting for a data packet, this function also sends "wakeup" packets to the data queues when it sends a command to the control queue. These wakup packets contain no data and are sent to all input-full queues and output-empty queues. In response, the component's default Datain(GetFull) and Dataout(GetEmpty) functions will check the control queue, as long you have specified the appropriate **tsaDatainCheckControl** or **tsaDataoutCheckControl** flag.

If a component-specific command is received, **tmalComInstanceConfig** is called directly to configure the component. The control command can be tsaCmdAcknowledge, **tsaCmdStatus**, or any component-specific command.

If the command is **tsaCmdStatus**, the component responds with a **tsaCmdAcknowledge** packet on the response queue to notify the application that the component is still alive.

The command argument includes one pointer field that can be used as the component designer sees fit. Since this pointer is weakly typed, it is a likely point of error for the application programmer. Some components provide a set of strongly typed configuration functions to avoid this problem. The command argument also contains a field called **retval**. On completion of the command sequence, this structure member contains the return value from the AL layer config function. The error returned by the OL layer config function will either be from the AL layer config function, or possibly an error telling you that the AL layer config function could not be dispatched. An example of this is encountered when you forget to install the control queues.

Since some applications do not need the configuration interface, the existence of the control queues is only checked when the default config function is called.

Multiprocess Considerations for InstanceConfig

Because a function cannot be called from another processor, the queue interface must be used to configure a component from another processor. In this case, the function of **tsaDefaultInstanceConfig** must be recreated in a function of the controlling processor to access the control queues of a component on the other processor. This involves possibly having to translate the address pointed to by the parameter field in the **tsaControlArgs** struct. Cache issues must also be taken into account.

AL InstanceConfig

```
tmLibappErr_t tmalCopyIOInstanceConfig(Int instance,ptsaControlArgs_t args){
   pInstVars_t   ivp   = (pInstVars_t) instance;
   Pointer       value = (Pointer)args->parameter;

   DP(("tmalCopyIOInstanceConfig(i:%x (p:%x))\n", ivp, ivp->parent));

   if (args == Null) return TMLIBAPP_ERR_INVALID_COMMAND;
   switch( (tmalCopyIOCommands_t)args->command ){
      case COPYIO_CHANGE_DELAY:
         ivp->delay = *(int*)value;
         DP((“CopyIO %x: change delay to %d\n”, ivp, ivp->delay);
         break;
      default:
         DP((“CopyIO %x: Command Unknown\n”, ivp));
         return TMLIBAPP_ERR_INVALID_COMMAND;
   }
   return TMLIBAPP_OK;
}
```

To prepare the component to accept configuration commands via the queue interface, a capabilities flag **tsaCapFlagsSupportControlQueue** must be added to the flags field in the **tsaDefaultCapabilities** struct, initialized in tmalCom.c. When this flag is set in the **tsaDefaultCapabilities** struct, the application must pass a **tsaControlDescriptor** to the component during **tmol***Com***InstanceSetup**, if it plans to configure the component by calling **tmol***Com***InstanceConfig**. When the application is using the AL layer, this function can be called directly to configure the component.

Component-specific commands must be **#define**d or **enum**ed as **tmal***Com***ConfigTypes_t** in the file tmalCom.h.

```
typedef enum {
    COPYIO_CHANGE_DELAY = tsaCmdUserBase;
} tmalCopyIOCommands_t;
```

If component-specific commands existed, a **tmal***Com***InstanceConfig** function must be provided. Then the **tmal***Com***InstanceConfig** function must be logged in the **tmal***Com***Funcs** table of the type **tsaDefaultFuncs**, in tmolCom.c.

A component can check its control queue by calling **Datain** or **Dataout** with the flag **tsaDatainCheckControlQueue** or **tsaDataoutCheckControlQueue**, respectively. **Datain** and **Dataout** check the control queue only if requested. This is the point of synchronization with the application.

Stop

OL Stop

```
tmLibappErr_t tmolCopyIOStop( Int instance ){
   pInstVars_t   ivp  = (pInstVars_t) instance;
   tmLibappErr_t rval = TMLIBAPP_OK;

   DP(("tmolCopyIOStop(i:%x (t:%x))\n", ivp, ivp->defInstVars->task));

   tsaCheckOpen (ivp->defInstVars, INST_OPEN_MASK,  INST_OPEN_MAGIC );
   tsaCheckSetup(ivp->defInstVars, INST_SETUP_MASK, INST_SETUP_MAGIC);

   rval = tsaDefaultStop(ivp->defInstVars);
   return rval;
}
```

The **tmol***Com***Stop** function is called by the application when using the OL layer to stop the data processing of a component. It simply calls **tsaDefaultStop**, which will then complete the rest of the stop sequence. It is essential to call **tsaDefaultStop** from any **tmol***Com***Stop** function, as it ensures the correct stopping of the component, by notifying its neighbors. **tsaDefaultStop** is used to stabilize the system after a component has been stopped.

Note
This function should not be called from an ISR.

Default Stop

```
extern tmLibappErr_t tsaDefaultStop(ptsaDefaultInstVar divp);
```

tsaDefaultStop takes the **tsaDefaultInstVars** struct as argument. It first calls **tmal***Com***Stop** to allow the component to first do internal cleanup on stop. It then sets the taskstatus of the component to **TS_STOP_REQUESTED**. **tsaDefaultStop** sends Wakeup packets to all input full queues and output empty queues, to break the component out of the blocking on a data queue. Therefore, the Datain/out, after receiving the Wakeup packet, can check the task status and notify the component task immediately. Having set the taskstatus to **TS_STOP_REQUESTED**, it waits for the component task to complete its stop sequence on the default instance variable **stopSemaphore**.

Default Task

When the component task sees that it has been requested to stop, either by **TMLIBAPP_STOP_REQUESTED** from a Datain/out call, or from having its component state set to **tsaCompStateStopRequested** in **tmal***Com***Stop**, it cleans up by returning packets outside the main loop, and exits to **tsa_default_task**. After returning from **tmal***Com***Start**, it sets the **receiverState** of all its input descriptors and **senderState** of all its output descriptors to stopped. It also flushes the input full queues to the empty queues and sends a **tsaCmdEndOfStream** packet down the output full queues.

Thus, the flushing of the full queues is always done by the receiver components. **tsaDefaultStopPin** can be used to flush output full queues. It then sets its **taskstatus** to **TS_NOTSTARTED**, releases the **stopSemaphore** so that **tsaDefaultStop** can complete and returns to the application. Lastly, it calls the completion function with the flag, **tsaCompletionFlagStop**, before suspending itself. The component task will be resumed on a subsequent call to **tmol***Com***Start**.

Calling **tsaDefaultStop** from within the task itself is asynchronous (i.e., **tsaDefaultStop** returns before the component is actually started). In that case, the application can wait for the completion function to be called with the flag **tsaCompletionFlagStop**.

AL Stop

```
tmLibappErr_t tmalCopyIOStop( Int instance ){
   pInstVars_t ivp = (pInstVars_t) instance;

   DP(("tmalCopyIOStop(%x)\n", instance));
   ivp->componentState = tsaCompStateStopRequested;
   return TMLIBAPP_OK;
}
```

In a task-based component, the **tmal***Com***Stop** function simply sets the component state to **tsaCompStateStopRequested**. When the task function **tmal***Com***Start** sees that the stop has been requested, it falls out of its processing loop. In a ISR-base component, however, **tmal***Com***Stop** must also expel internally held packets into the queues.

This function is called first in the **tsaDefaultStop**, before the application begins the stop sequence. In both task-based and ISR-based components, the function must not wait for the component task (because stop could have been called from the task itself). In that case, the **tmal***Com***Stop** function must terminate before the task can exit its processing loop and complete the stop sequence.

Close

OL Close

```
tmLibappErr_t tmolCopyIOClose( Int instance ){
   pInstVars_t   ivp  = (pInstVars_t) instance;
   tmLibappErr_t rval = TMLIBAPP_OK;

   DP(("tmolCopyIOClose(i:%x, (t:%x))\n", ivp, ivp->defInstVars->task));

   tsaCheckOpen(ivp->defInstVars, INST_OPEN_MASK, INST_OPEN_MAGIC);

   rval = tsaDefaultClose(ivp->defInstVars);
   free(ivp->setup);
   free(ivp);

   return rval;
}
```

The **tmol***Com***Close** function is used by the application when using the OL layer to release an instance created by **tmol***Com***Open**. It calls **tsaDefaultClose** to free memory allocated by **tsaDefaultOpen** and frees all other memory allocated during **tmol***Com***Open**.

Note that this function should not be called from an ISR.

Default Close

```
extern tmLibappErr_t tsaDefaultClose(ptsaDefaultInstVar divp);
```

tsaDefaultClose takes the **tsaDefaultInstVars** struct as argument. It first calls **tmal***Com*-**Close** to allow the AL layer to free all the memory it allocated in **tmal***Com***Open**. It then makes sure that the component's stop sequence has completed, by checking if the taskstatus is **TS_NOTSTARTED**. If not, it waits until it is. This is necessary because a component instance must do the necessary cleanup, such as expelling packets and releasing handles to hardware, before being closed. Finally, it destroys the semaphores and the component task, and frees all the memory that was allocated in **tsaDefaultOpen**.

AL Close

```
tmLibappErr_t tmalCopyIOClose( Int instance ){
   pInstVars_t   ivp  = (pInstVars_t) instance;
   tmLibappErr_t rval = TMLIBAPP_OK;

   DP(("tmalCopyIOClose(i:%x (p:%x))\n", ivp, ivp->parent));
   def_cap.numCurrentInstances--;
   free(ivp->setup->defaultSetup);
   free(ivp->setup);
   free(ivp);
   return rval;
}
```

In the **tmal***Com***Close** function, it decrements the **numCurrentInstances** field in the **tsaDefaultCapabilities** struct. It then frees all memory allocated during **tmal***Com***Open**.

ProcessData

The **ProcessData** function performs OS-independent data processing. It only exists in the AL layer and is directly called by the application when using the AL layer of a component in non-streaming mode. Components can have more than one **ProcessData** function, which can be called something that more accurately describes its functionality. For example, CopyIO's **ProcessData** functions is called **tmalCopyIOCopyPacket**.

AL ProcessData

```
tmLibappErr_t tmalCopyIOCopyPacket(
   Int instance, tmAvPacket_t * inpacket, tmAvPacket_t *outpacket
){
   pInstVars_t    ivp  = (pInstVars_t) instance;
   tmLibappErr_t  err  = TMLIBAPP_OK;
   int            i;

   DP(("tmalCopyIOCopyPacket(i:%x (p:%x))\n", ivp, ivp->parent));

   ivp->copyInProgress = True;

   if( outpacket->allocatedBuffers < inpacket->allocatedBuffers ){
      ivp->copyInProgress = False;
      return CP_ERR_ALLOCATED_BUFFERS;
   }
   for( i=0; i<outpacket->allocatedBuffers; i++ ){
      if(outpacket->buffers[i].bufSize < inpacket->buffers[i].bufSize) {
         ivp->copyInProgress = False;
         return CP_ERR_BUFSIZE;
      }
   }
   outpacket->header->flags        = inpacket->header->flags;
   outpacket->header->userSender   = inpacket->header->userSender;
   outpacket->header->userReceiver = inpacket->header->userReceiver;
   outpacket->header->userPointer  = inpacket->header->userPointer;
   outpacket->header->time.ticks   = inpacket->header->time.ticks;
   outpacket->header->time.hiTicks = inpacket->header->time.hiTicks;

   outpacket->buffersInUse         = inpacket->buffersInUse;

   for( i=0; i<outpacket->allocatedBuffers; i++ ){
      memcpy(outpacket->buffers[i].data,
             inpacket ->buffers[i].data,
             inpacket ->buffers[i].dataSize );
      outpacket->buffers[i].dataSize = inpacket->buffers[i].dataSize;
   }

   ivp->copyInProgress = False;
   return err;
}
```

tmal*Com***ProcessData** and its counterparts embody the AL non-streaming operations of a streaming component. Unlike **tmal***Com***Start**, it is not responsible for acquiring data to process. Instead, it is given data to process by the application, and the processed result is returned to the application through a parameter.

Summary of Design Models

The following is a summary of the choices in designing a component.

Streaming vs. Non-Streaming

Components can be streaming or non-streaming. If a component streams data, as do most digital signal processing (DSP) components, an OL layer must be part of the component to use queues to stream data. This component would be a TSSA component. Alternatively, components that do not stream data, as for most functional libraries, need not have an OL layer. Applications would access components through function calls in the AL layer only. This component would be a TSA component, not a TSSA component.

Data Processing

A TSSA component can implement its data processing with or without dependency on an operating system. The core routines of signal processing components are particularly well suited to implementation without explicit reliance on an OS. In these cases, the serious data processing is confined to the AL layer, and the operating system dependencies are isolated to a higher level, through the standard set of TSSA callback functions.

But sometimes the signal processing is not so clearly separable from the operating system. A component that parses and maintains a database is an example of this. Semaphores and other operating system features can be an integral part of the component's basic implementation. In this case, the line between the AL and the OL layer becomes blurred. Because of the dependency on the OS, the component will present an OL layer interface to the outside world. The author can dispense with the AL layer completely, or if operating system independence is a goal of the author, component-specific callback functions can be introduced allowing the OL layer to provide functionality to the AL layer.

Pull vs. Push Model

Streaming components can support both pull and push models. The pull model implies that an autonomous component pulls in data for processing from queues. The label "push model" is applied to situations where an application pushes data into a component to be processed.

When a streaming component pulls in data to process, it waits for data to be streamed in from the queues. This happens when the application uses the OL layer of the component with **tmol***Com***Start** and **tmol***Com***Stop**.

However, when the application pushes data in to the component to be processed, it is using the AL layer, specifically the component's **tmal***Com***ProcessData** function. A component can have more than one **tmal***Com***ProcessData** function.

Task-Based vs. ISR

Components that process data should be task-based. Components that access hardware (through device libraries) are likely to be implemented as an Interrupt Service Routine (ISR). These components might or might not include a task. An ISR-based component waits for a signal from the hardware before executing one "loop" of its processing. Digitizers and renderers are examples of ISR-based components.

Component Packages

TSA-compliant components are composed of header files, library files, and example files.

- Header files of the components are found in $(TAS)/include/ under the names tmolCom.h and tmalCom.h for streaming components, and tsaCom.h for non-streaming components. For applications using the OL layer of a streaming component, tmalCom.h is already included in tmolCom.h.
- The files that make up each component library are found in the $(TAS)/lib/Com/ directory.

  Simple components have in that directory tmolCom.c and tmalCom.c. More complex components have the files that compose of the AL layer in a tmal sub-directory. These components have tmolCom.c in $(TAS)/lib/Com/, while the tmal sub-directory contains tmalCom.c and other files.
- Each component found in $(TAS)/lib/ has a corresponding example that shows how to use it in a directory under $(TAS)/examples/. Examples for non-streaming components are named exCom (e.g. **ex2D**) or extsaCom.

  Examples for streaming components are prefixed with exal, for an example using the AL layer, and exol, for an example using the OL layer. exal or exol are followed by a component name, or a name describing the functionality of the example.

  If **exal** or **exol** is followed by a component name, then it is mainly demonstrating the use of the component, possibly with other components. For example, **exolVrendVO** demonstrates the use of **VrendVO** with **VdigVI**.

  If **exal** or **exol** is followed by something other than a component name, then it describes the functionality of the example. For example, **exolFileIO** demonstrates the use of **Fread** and **Fwrite** doing file I/O.

Chapter 11

TSSA Design Details

Introduction

This chapter presents more advanced and detailed concepts of the TSSA architecture that go beyond the basic ideas explained in the previous chapter. These concepts fall into two parts, those pertaining to component design and those pertaining to application design.

The first part of this chapter demonstrates various choices in component design, using code from applicable components. It addresses these concepts:

- ISR components.
- In-place components.
- Changing formats from inside a component.
- Components that wait on multiple queues.

The second part of this chapter presents several practical uses of TSSA at the application level. It addresses these concepts:

- Synchronized stop.
- Changing formats from the application.
- Reconnecting sender and receiver components.

The previous four chapters (beginning with Chapter 7, *TSSA Essentials*) are essential background for this chapter.

Component Design Details

ISR Components

ArendAO is an ISR component because it accesses hardware by rendering audio, and is therefore driven by interrupts from the Audio Out device. Using TSSA, there are two steps required to create an ISR component: setting **createNoTask** and providing a **ReceiverSetupFormat** function.

createNoTask

```
tmLibappErr_t tmolArendAOOpen ( Int *instance ){
   tmLibappErr_t rval;

   DP(("tmolArendAOOpen(): "));
   InstVars.setup = (ptmolArendAOInstanceSetup_t) calloc(1,
   sizeof(tmolArendAOInstanceSetup_t) );
   if( !InstVars.setup ) return (TMLIBAPP_ERR_MEMALLOC_FAILED);
   rval = tsaDefaultOpen( &(InstVars.defInstVars), INST_OPEN_MAGIC,
                          &arendTmalFunc );
   if (rval != TMLIBAPP_OK) {
      free (InstVars.setup);
      return rval;
   }
   strcpy(InstVars.defInstVars->instSetup->taskName, "ArendAO");
   InstVars.defInstVars->instSetup->createNoTask = True;
   InstVars.setup->defaultSetup =
   InstVars.defInstVars->instSetup;
/* setup the ArendAO specific things */
   InstVars.format                = apfStereo16;
   InstVars.srate                 = 44100.0;
   InstVars.started               = False;
   InstVars.setup->output         = InstVars.output          = aaaNone;
   InstVars.setup->maxBufferSize  = InstVars.maxBufferSize   = 1024;
   InstVars.setup->operationalMode = InstVars.operationalMode = AR_MODE_RAW;

   *instance = (UInt32) &InstVars;
   return TMLIBAPP_OK;
}
```

tmolArendAOOpen is similar to **tmolCopyIOOpen**, except that **ArendAO** sets **createNoTask** in the default instance setup struct to true. This variable affects the way the component is started by **tsaDefaultStart** and therefore must be set before **tmol***Com***Start** is called. When it is set to true, **tsaDefaultStart** will directly call **tmal***Com***Start**, instead of starting a task, as in CopyIO. Besides this small part in **tmol***Com***Open**, the other OL functions of an ISR component should not be much different from those of a task-based component.

tmal*Com*ReceiverFormatSetup

Components that need a specific format before startup must provide a **tmal***Com***ReceiverFormatSetup** function. **tmal***Com***ReceiverFormatSetup** sets up the format of a receiver component and is part of the instance setup of the component, although separate from its **tmal***Com***InstanceSetup**. Often, but not always, this function exists for ISR receiver components because most hardware devices need to know the format before startup. Other receiver components that need the format before startup should also provide this function. This function is needed for receiver components, because, unlike sender components, they have no control over the format of their input streams. A sender component can specify the format of its output streams according to the format of its input

stream, or from the application through instance config. When **tmal*Com*ReceiverFormatSetup** exists, it must be entered into the default capabilities of the component.

```
static tsaDefaultCapabilities_t defaultCaps = {
   ccAudioRenderer,                 /* componentClass        */
   {1, 1, 0},                       /* version               */
   tsaCapFlagsCopybackDatain,       /* capabilityFlags       */
   TEXT_MEMORY_REQUIREMENT,         /* textmemoryRequirement */
   DATA_MEMORY_REQUIREMENT,         /* datamemoryRequirement */
   PROCESSOR_REQUIREMENT,           /* processorRequirement  */
   1,                               /* numSupportedInstances */
   0,                               /* numCurrentInstances   */
   ARENDAO_NUMBER_OF_INPUTS,        /* numberOfInputs        */
   parFormat,                       /* inputFormats          */
   ARENDAO_NUMBER_OF_OUTPUTS,       /* numberOfOutputs       */
   Null,                            /* outputFormats         */
   tmalArendAOReceiverFormatSetup   /* receiverFormatSetup   */
};
```

tmal*Com*ReceiverFormatSetup is called whenever the format of an input **InOutDescriptor** is known, by **tsaDefaultInstallFormat**. **tsaDefaultInstallFormat** can be called in one of two places. When the application calls **tsaDefaultInOutDescriptorCreate** with a format specified, or when the sender component figures out or changes the format of its output stream. The way that a sender component specifies the format will be described in *Changing Formats in Components* starting on page 100.

Setting up the Format for the Component

```
static tmLibappErr_t tmalArendAOReceiverFormatSetup(
   UInt32 inputIndex, Pointer format
){
   tmLibdevErr_t    rval = TMLIBAPP_OK;
   ptmAudioFormat_t newFormat = (ptmAudioFormat_t)format;

   DP((“ArendAOReceiverFormatSetup\n”));
   *InstVars.format = *newFormat;

   if( !InstVars.formatInstalled ){
      switch( newFormat->dataSubtype ){
         case apfMono16:
            InstVars.bytesPerSample = sizeof(Int16);
            break;
         case apfStereo16:
            InstVars.bytesPerSample = 2 * sizeof(Int16);
            break;
        ... /* more data subtypes */
         default:
            DP(("aoStartFunc: got unsupported format %x\n",
               AudioFormat.dataSubtype));
            InstVars.bytesPerSample = 2 * sizeof(Int16);
            break;
      }
   }
   if( InstVars.setup ) rval = aoStartFunc();
   InstVars.formatInstalled = True;
   return rval;
}
```


tmal*Com***ReceiverFormatSetup** takes as arguments the index into the array of input descriptors, and the format to be installed in the component. **tmalArendAOReceiver-FormatSetup** sets up an AL instance variable, **bytesPerSample**, according to the format.

Finally, it calls **devStartFunc**, if **tmal***Com***InstanceSetup** has already been called. Correspondingly, **tmal***Com***InstanceSetup** calls **devStartFunc** if **tmal***Com***ReceiverFormatSetup** has already been called.

```
tmLibappErr_t
tmalArendAOInstanceSetup(Int instance, tmalArendAOInstanceSetup_t *setup){
   ... /* set up ArendAO */
   if( InstVars.formatInstalled ) rval = aoStartFunc();
   InstaVars.setup = True;
   return rval;
}
```

Setting up and Starting the Hardware Device

aoStartFunc sets up and starts the Audio Out device according to the format. Non-ISR receiver components that need the format for setup should also have a **tmal***Com***Receiver-FormatSetup** function. However, these components need not have a **devStartFunc** function, because they do not need to set up any hardware device.

```
static tmLibappErr_t aoStartFunc( void ){
   aoInstanceSetup_t   ao;
   tmLibappErr_t       rval = TMLIBAPP_OK;
   L1_DP(("aoStartFunc\n"));

   ao.interruptPriority  = intPRIO_3;
   ao.audioTypeFormat    = InstVars.format->dataType;
   ao.audioSubtypeFormat = InstVars.format->dataSubtype;
   ao.underrunEnable     = False;
   ao.hbeEnable          = False;
   ao.buf1emptyEnable    = True;
   ao.buf2emptyEnable    = True;

   if (InstVars.datainFunc) ao.isr = arISR_Streaming;
   else                     ao.isr = arISR_nonStream;

   ao.base1   = pZbuf;            /* setup for silence */
   ao.base2   = pZbuf;
   ao.size    = InstVars.maxBufferSize / InstVars.bytesPerSample;
   ao.sRate   = InstVars.sRate;
   ao.output  = InstVars.output;

   if( rval = aoInstanceSetup(AO_instance, &ao) ){
      DP(("AO returned %x\n", rval));
      return rval;
   }
   if( rval = aoStart(AO_instance) ){
      DP(("aoStart returned %#x\n", rval));
      return rval;
   }
   return rval;
}
```

The **aoStartFunc** above fills an AO instance setup structure and passes it to **aoInstance-Setup**. Then it calls **aoStart** to start the AO hardware device.

In-Place Components

Sometimes, a component does not want to circulate the data memory in a packet between its input component and output component separately. In other words, instead of processing the data from an input packet and putting the results into separate data memory in an output packet, it passes the pointer to the data memory from its input to output. In TSSA terminology, this is called processing data in place. In-place processing is transparent to the application and default datain/out behave accordingly if the component specifies **tsaCapFlagsInPlace** in its default capabilities structure. CopyInPlace is such a component.

tsaCapFlagsInPlace

```
static tsaDefaultCapabilities_t def_cap = {
   ccGenericIn,                          /* component class       */
   {1,1,0},                              /* version               */
   tsaCapFlagsSupportsControlQueue |     /* capabilityFlags       */
   tsaCapFlagsInPlace,
   TEXT_MEMORY_REQUIREMENT,
   DATA_MEMORY_REQUIREMENT,
   PROCESSOR_REQUIREMENT;
   NUM_SUPPORTED_INSTANCES,
   0,                                    /* numCurrentInstances   */
   COPYIO_NUMBER_OF_INPUTS,
   id_format,                            /* inputFormats          */
   COPYIO_NUMBER_OF_OUTPUTS,
   od_format,                            /* outputFormats         */
   Null                                  /* receiverFormatSetup   */
};
```

When **tsaCapFlagsInPlace** is specified, the component is expected to send data packets from the input full queue directly to the output full queue. The input and output queues must have corresponding IDs in the component's list of inputs and outputs. An in-place component also never accesses any empty queues. The corresponding input and output components share one empty queue while the in-place component is running. Queues are rearranged in the defaults during **tsaDefaultStart** and **tsaDefaultStop** so the output empty queue is connected directly to the input component, thus bypassing the in-place component. This scheme avoids the extra overhead involved if the in-place component is needlessly accessing empty queues.

Changing Formats in Components

When two components are connected, only the sender or the application can change the format of the connection. The receiver must respond to the format change accordingly.

Sender: Initiating Format Change

To change the format of an output pin while processing data, a sender component must call the progress function with the flag **tsaProgressFlagChangeFormat**. The progress code should contain the index of the output descriptor to change, and the description should contain the desired format. The default layer then responds by installing the new format in the **InOutDescriptor**. During the installation, **tmal***Com***ReceiverFormatSetup** of the receiver is called. If an error occurred during the installation of the new format, the default layer will call the error function with an **NonFatal** flag, and the error as the error code. If there was no error in the installation of the format, then the sender component effectively changed the format of the output connection.

```
    tsaProgressArgs_t prog_args;

    prog_args.progressCode = 0;
prog_args.description  = (Pointer)&format;
rval = ivp->progressFunc(ivp->parentId, tsaProgressFlagChangeFormat,
&prog_args);
```

In the example above, the output index is 0 and the format is the pointer to a statically allocated format.

Note: The sender component should not initiate a format change within an interrupt service routine.

Receiver: Responding to Format Change

The receiver is notified of a change in format when getting a new packet through its input pin in which the format has been changed by the sender. **Datain(GetFull)** will return an error **TMLIBAPP_NEW_FORMAT** with the new packet. From this error, the receiver can then process the packet accordingly. Note that the receiver also receives this error from **Datain(GetFull)** on the first packet ever sent to this input pin. For receiver components that do not need to do special processing when a new format is received, it can just set the error value back to **TMLIBAPP_OK** and continue processing. An example of this is in CopyIO from above. The following code ignores the **TMLIBAPP_NEW_FORMAT** error, but breaks from the processing loop with another error.

```
err = ivp->datainFunc(ivp->parent,tsaDatainGetFull|tsaDatainWait,&di_args);
ivp->inPacket = (ptmAvPacket_t)di_args.packet;
if( err == TMLIBAPP_NEW_FORMAT ) err = TMLIBAPP_OK;
else if( err ) break;
```

Waiting on Multiple Input Queues with waitSemaphore

Receiver components can wait on multiple input pins by setting up the **InOutDescriptors** of its input pins to do so. **Datain(GetFull)** will wait on a semaphore instead of individual queues when the semaphore exists. Correspondingly, **Dataout(PutFull)** will release the semaphore if it exists. This semaphore is stored in each **InOutDescriptor** as **waitSemaphore**. The following is an example of a component that waits on multiple input queues.

Setting Up Inputs with waitSemaphore

```
tmLibappErr_t tmolAmixSimpleInstanceSetup(
   Int Instance, ptmolAmixSimpleInstanceSetup_t setup
){
   ... /* set up AmixSimple */

   if( rval = tmosSemaphoreCreate ("AMIX",tmosSemaphoreFlagsStandard,
       0,&waitSemaphore) ) return rval;

   for( i=0; i<AMIXSIMPLE_NUMBER_OF_INPUTS; i++) {
      if( ivp->setup->defaultSetup->inputDescriptors[i])
         ivp->setup->defaultSetup->inputDescriptors[i]->waitSemaphore
         = waitSemaphore;
   }
   ...
}
```

To set up the **InOutDescriptors**, the component creates the wait semaphore and stores it in the **InOutDescriptors** of all of its input pins. This is done in **tmol***Com***InstanceSetup** after calling **tsaDefaultInstanceSetup**.

Using Datain(GetFull) with waitSemaphore

```
tmLibappErr_t tmalComStart( Int Instance ){
   tsaDatainArgs_t diArg;

   ... /* initialize other variables */

   ivp->componentState = tsaCompStateRunning;

   while( ivp->componentState == tsaCompStateRunning ){
      diArg.inputId = 0;
      diArg.timeout = 0;
      err =
      ivp->datainFunc(ivp->parentId,tsaDatainGetFull|tsaDatainWait,&diArg);

      ivp->inpacket = diArg.packet;

      switch(err) {
         case TMLIBAPP_OK:                /* this queue contains packet */
            break;
            ...
         case TMLIBAPP_QUEUE_EMPTY:       /* check other queues */
            for( i=1; i<COM_NUMBER_OF_INPUTS; i++ ){
               diArg.inputId = i;
               diArg.timeout = 0;
               err = ivp->datainFunc(ivp->parentId, tsaDatainGetFull,
               &diArg); /* default is NoWait */
               ivp->inpacket = diArg.packet;
               if(err == TMLIBAPP_OK) break;
            }
            break;
         ...
         default:          /* break out of while loop for any other errors */
            goto Cleanup;
      }
      ProcessPacket(diArg.packet);
      ...
   }
   Cleanup:
   ...
}
```

In **tmal*Com*Start**, if waiting on multiple queues, **Datain(GetFull)** can return an error, **TMLIBAPP_QUEUE_EMPTY**, even if the flag specified **tsaDatainWait**. In this case, **TMLIBAPP_QUEUE_EMPTY** does not indicate that the timeout occurred; instead it indicates that the queue specified in the **inputId** is not the queue that received a packet. Further calls to **Datain(GetFull)** with subsequent **inputIds** are necessary before finding the queue that received the packet. The packet is retrieved when **Datain(GetFull)** return **TMLIBAPP_OK**.

In the example above, the component waits on input queue 0 until a packet is received on one of the **COM_NUMBER_OF_INPUTS** queues. If there is no error from **Datain(GetFull)**, then the packet was received on input queue 0, and the packet can be processed immediately. However, if the error from **Datain(GetFull)** is **TMLIBAPP_QUEUE_EMPTY**, then the packet was received on one of the other queues. Starting with input queue 1,

calls to **Datain(GetFull)** can be made with no wait, until the packet is found; i.e., when a **TMLIBAPP_OK** is returned. Then, the input packet can be processed.

In the event of a stop request from the application, the **Datain(GetFull)** will return a **TMLIBAPP_STOP_REQUESTED**. When the component receives this or other fatal errors, it must break out of the while loop in which the component is running. The TSSA default layer will take care of flushing the queues and reinitializing the **waitSemaphore** of this component as part of the stop process.

Calculating Memory Requirements

You must define **TEXT_MEMORY_REQUIREMENT**, **DATA_MEMORY_REQUIREMENT**, and **PROCESSOR_REQUIREMENT** and give them values before releasing components in both tmolCom.c and tmalCom.c. The application can retrieve the memory and processor requirements using the component's **GetCapabilities** function. When **GetCapabilities** is called in the OL layer, the AL layer requirements are automatically added.

- To calculate **TEXT_MEMORY_REQUIREMENT**, use **tmsize** for all object files in the each layer (OL and AL) and set **TEXT_MEMORY_REQUIREMENT** to the number reported for text.
- To calculate **DATA_MEMORY_REQUIREMENT**, use **tmsize** for all object files in the each layer and set **DATA_MEMORY_REQUIREMENT** to the sum of the numbers reported for data, data1, and bss, plus any memory allocated without the use of the **memalloc** call-back function. These allocated memory are usually the instance variables structure and its elements.
- **PROCESSOR_REQUIREMENT** is not used currently, so set to 0.

Example

The following is the output of **tmsize** on tmolCopyIO.o, which is the only object file in the OL layer for CopyIO, and the resulting definitions of the memory requirements in tmolCopyIO.c.

```
text      data      data1     bss       dec       hex
1263      36        5         4         1308      0x51C
```

```
#define TEXT_MEMORY_REQUIREMENT 1263
#define DATA_MEMORY_REQUIREMENT 37 +                                \
                                sizeof(InstVars_t) +                \
                                sizeof(tmolCopyIOInstanceSetup_t) + \
                                sizeof(tsaDefaultInstVar_t) +       \
                                sizeof(tsaDefaultInstanceSetup_t)
#define PROCESSOR_REQUIREMENT   0
```


Likewise, the following is the output of **tmsize** on tmalCopyIO.o, which is the only object file in the AL layer for CopyIO, and the resulting definitions of the memory requirements in tmalCopyIO.c.

```
text      data      data1     bss       dec       hex
3166      88        1         0         3255      0xCB7
```

```
#define TEXT_MEMORY_REQUIREMENT 3166
#define DATA_MEMORY_REQUIREMENT 89 +                                  \
                                sizeof(InstVars_t) +                  \
                                sizeof(tmalCopyIOInstanceSetup_t) + \
                                sizeof(tsaDefaultInstanceSetup_t)
#define PROCESSOR_REQUIREMENT   0
```

Application Design Details

Using non-TSSA components

The easiest way to use a non-TSSA component with other TSSA components is to wrap it in a TSSA wrapper. Then you are connecting two TSSA components. If you were to connect a TSSA component to a non-TSSA component, you would need to know much more about the inner workings of a TSSA component. The bulk of these "inner workings" is coded in the files tsaDefaults.c and tsaFormats.c. These two files are provided as source, so you can study the details.

Two specific issues are likely to cause trouble unless you know about them.

1. Creation of the connections between components.

   These connections are embodied in I/O descriptors. The set of default functions includes a function to create such a descriptor, but this creation function requires the capabilities structure for each component. When you use a non-TSSA component, you must construct such a capabilities structure to complete the connection.

2. The way data is passed through the queues.

   A useful set of interface functions are reproduced here:

```
extern UInt32 qDataOutPutFull(UInt32 fullQ, ptmAvPacket_t packet){
   UInt32   msg_buf[4];

   packet->buffersInUse = 1;
   msg_buf[0]           = (UInt32) packet;
   msg_buf[1]           = tsaCmdDataPacket;
   return q_send(fullQ, msg_buf);
}
```

Notice how **msg_buf[1]** as passed through the queue contains a command identifying this as a data packet.

```
extern UInt32 qDataOutGetEmpty(
   UInt32 emptyQ,
   UInt32 waitFlag,
   ptmAvPacket_t *packet
){
   UInt32    err, msg_buf[4];

   msg_buf[1] = tsaCmdAcknowledge;              /* something not data */
   while (msg_buf[1] != tsaCmdDataPacket) {
   /* throw away pause and ack packets */
      if( err = q_receive(emptyQ,waitFlag,0,msg_buf) ){
         *packet = (ptmAvPacket_t) msg_buf[0];
         return (err);
      }
   }
   *packet = (ptmAvPacket_t) msg_buf[0];
   return (0);
}
```

Here, non-data packets are thrown away.

```
extern UInt32 qDataInPutEmpty(UInt32 emptyQ, ptmAvPacket_t packet){
   UInt32   msg_buf[4];

   if( packet->header->format )
      tsaFormatRelease(packet->header->format);
   msg_buf[0] = (UInt32) packet;
   msg_buf[1] = tsaCmdDataPacket;
   return q_send(emptyQ, msg_buf);
}
```

```
extern UInt32 qDataInGetFull(
   UInt32 fullQ,
   UInt32 waitFlag,
   ptmAvPacket_t * packet)
){
   UInt32   err, msg_buf[4];

   msg_buf[1] = tsaCmdAcknowledge;            /* something not data */
   while( msg_buf[1] != tsaCmdDataPacket ){
      /* throw away pause and ack packets */
      if( err = q_receive(fullQ,waitFlag,0,msg_buf) ){
         *packet = (ptmAvPacket_t) msg_buf[0];
         return (err);
      }
   }
   *packet = (ptmAvPacket_t) msg_buf[0];
   return (0);
}
```

Synchronized Stop

In a streaming architecture, it is often desirable to synchronize stopping of connected components. “Synchronized stop” means that given two connected components, the receiver does not stop until it has processed all the packets the sender sent before it

stopped. TSSA provides a mechanism for synchronizing stop that uses the progress callback function.

End of Stream

When the sender of a connection is being stopped, the default layer will send an end-of-stream packet with the **tsaCmdEndOfStream** command. This part of the stop process does not require any participation from the application. When the receiver receives this **tsaCmdEndOfStream** packet, it calls the progress callback function with the flag **tsaProgressFlagEndOfStream**. This is also done in **tsaDefaultDatainFunction**.

However, if you want your application to make use of this information, you must provide a progress callback function to catch the **tsaProgressFlagEndOfStream** flag. At that point, the application may choose to stop the receiver.

The example following comes from exolCopyIO, where Fread is connected to CopyIO, which is connected to Fwrite. In this example, we focus on the Fread-CopyIO connection (that is, Fread is the sender and CopyIO is the receiver). To synchronize the stopping of Fread and CopyIO, the application must provide a progress function when setting up CopyIO.

```
CopyIOSetup->defaultSetup->progressFunc = lCopyIOProgFunc;
```

This progress function must catch the **tsaProgressFlagEndOfStream** flag:

```
tmLibappErr_t
lCopyIOProgFunc( Int inst, UInt32 flags, ptsaProgressArgs_t args ){
   printf( "CopyIO Progress: instance 0x%x flags 0x%x code 0x%x\n", inst,
           flags, args->progressCode );
   switch( flags ){
      case tsaProgresFlagEndOfStream:
         printf("Releasing CopyIOSema\n");
         tmosSemaphoreV(CopyIOSema);
         break;
      default:
         break;
   }
   return TMLIBAPP_OK;
}
```

Here, this progress function releases the semaphore, **CopyIOSema**, when it detects the **tsaProgressFlagEndOfStream** flag. At that moment, the main function, having stopped Fread, is waiting on **CopyIOSema** before stopping CopyIO. The same concept synchronizes the stopping of CopyIO and Fwrite.

Changing Formats from the Application

When the application determines, from user input or other means, the need to change the format of a connection, it should proceed according to whether the sender component supports format changes with its instance configuration function.

Using tsaDefaultInstallFormat

The application calls **tsaDefaultInstanceFormat** with the InOutDescriptor and the new format as arguments. **tsaDefaultInstanceFormat** informs the receiver by calling its **tmal***Com***ReceiverFormatSetup** callback function, which should store the new format for the receiver. See *tmalComReceiverFormatSetup* on page 97. The receiver is notified, but the sender is not yet notified of the format change. A separate call to **tmolComInstanceSetup** or **tmolComInstanceConfig** is required before the sender starts producing packages with the new format. After that call, the two components can be restarted. The following code comes from exolVrendVO.

```
/* set up new format */
   if( !plainBuffer & fieldBased )
      digitizer_format.desription = vdfFieldInFrame;
   else if( !plainBuffer & !fieldBased )
      digitizer_format.description = vdfInterlaced;
   else if( plainBuffer & fieldBased )
      digitizer_format.description = vdfFieldInField;

/* call tsaDefaultInstanceFormat */
   rval = tsaDefaultInstallFormat(
   digitizer_inst_setup->defaultSetup->outputDescriptors[VDIGVI_MAIN_OUTPUT],
   (ptmAvFormat_t)&digitizer_format );

/* notify sender */
   digitizer_inst_setup->fieldBased = fieldBased;
   digitizer_inst_setup->interlaced = !plainBuffer;
   rval = tmolVdigVIInstanceSetup( digitizerInstance, digitizer_inst_setup );
```

Note: An application must not call **tsaDefaultInstallFormat** from within an interrupt service routine.

Using tmolComInstanceConfig

The most convenient way for an application change the format in a connection is for the sender component to provide a **COM_CHANGE_FORMAT** command for instance configuration. If the sender provides a **COM_CHANGE_FORMAT** configuration command, the application need only call **tmolComInstanceConfig** with the **COM_CHANGE_FORMAT** command and with the new format as the parameter. The preceding code will become the following code:

```
/* set up new format */
   if( !plainBuffer & fieldBased )
      digitizer_format.desription = vdfFieldInFrame;
   else if( !plainBuffer & !fieldBased )
      digitizer_format.description = vdfInterlaced;
   else if( plainBuffer & fieldBased )
      digitizer_format.description = vdfFieldInField;

/* call tmolVdigVIInstanceConfig */
   cargs.command = VDIGVI_CHANGE_FORMAT;
   cargs.parameter = (Pointer)&digitizer_format;
   rval = tmolVdigVIInstanceConfig(digitizerInstance,tsaControlWait,&cargs);
```


To provide a **COM_CHANGE_FORMAT** command for instance configuration, the **tmal***Com***InstanceConfig** function receiving this command should call the progress function with **tsaProgressFlagChangeFormat** as mentioned above. See *Sender: Initiating Format Change* on page 101. This is a shortcut for the application and allows it to avoid calling **tsaDefaultInstanceFormat** and reconfiguring the sender, and shortens this application-initiated format change into one quick step.

Reconnecting Components

Because the only sender can affect the format of a connection, while the receiver can only accept or refuse the format, the reconnecting of sender and receivers must be treated differently.

Reconnecting Sender

To reconnect a sender, the application need only call **tsaDefaultSenderReconnect**. (The sender must be stopped.) At that point, the application can specify a new format for the connection, or it can use the existing format by passing **Null** for the format.

In the following example, **iod** originally connects Fread and Fwrite, using a generic format. The application wants to replace Fread with AdigAI as the sender, and install **audioFormat** as the new format.

```
rval = tsaDefaultSenderReconnect( iod, FWRITE_MAIN_INPUT,
                                  AdigAIInst->defaultCapabilities,
                                  (ptmAvFormat_t)&audioFormat );
```

This works because Fwrite will accept the new audio format.

Reconnecting Receiver

To reconnect a receiver, the application need only call **tsaDefaultReceiverReconnect**. (The receiver must be stopped.) The new receiver's possible formats will be negotiated against the existing format. In the following example, **iod** originally connects Fread and Fwrite using a generic format. The application wants to replace Fwrite with ArendAO as the receiver. Because ArendAO expects packets in an audio format, and does not accept the existing generic format, the format of the connection must be changed to **audioFormat** before the call to **tsaDefaultReceiverReconnect**.

```
rval = tsaDefaultInstallFormat( iod, (ptmAvFormat_t)&audioFormat );
rval = tsaDefaultReceiverReconnect( iod, FREAD_MAIN_OUTPUT,
                                    ArendAOInst->defaultCapabilities);
```

To reconnect the receiver and install a new format, the old receiver must be able to handle the new format. In this example, Fwrite can handle **audioFormat** and therefore, the format can be changed before reconnecting the receiver to ArendAO.

Chapter 12

TSSA Compliance

Introduction

This chapter provides a TSSA compliance checklist for TSSA component designers. The checklist addresses items to be examined in the header files, in the library code, and the example and test code of the component. Examples of code are given where applicable, with *Com* representing the component, italicized code allowing component-specific variations, and ellipses (…) allowing component-specific additions. This TSSA compliance checklist is currently a guideline, but will develop into a TSSA compliance test suite in the near future. Presently, some deviations from the checklist may be discussed. Note that the checklist does not apply to non-streaming (TSA) components.

Header Files

Every TSSA component must export a tmol*Com*.h and a tmal*Com*.h (*Com* representing a name for the component) to be included by applications that use the component. Each tmol*Com*.h and tmal*Com*.h must at least adhere to the points following for the component to be considered TSSA-compliant. Finally, the component error base must be defined after consulting tmLibappErr.h to avoid having the same error codes as other existing TriMedia components.

tmol*Com*.h

- tmol*Com*.h must include tmal*Com*.h:

```
#include <tmalCom.h>
```

- In the type definition of the structure **tmol*Com*Capabilities_t**, the first member must be a pointer to a structure of type **tsaDefaultCapabilities_t**:

```
typedef struct {
   ptsaDefaultCapabilities_t defaultCapabilities;
   ...
} tmolComCapabilities_t, ptmolComCapabilities_t;
```

- In the type definition of the structure **tmol*Com*InstanceSetup_t**, the first member must be a pointer to a structure of type **tsaDefaultInstanceSetup_t**

```
typedef struct {
   ptsaDefaultInstanceSetup_t defaultSetup;
   ...
} tmolComInstanceSetup_t, ptmolComInstanceSetup_t;
```

- Eight basic TSSA OL functions must be declared with appropriate arguments:

  tmol*Com*GetCapabilities

```
extern tmLibappErr_t tmolComGetCapabilities(
ptmolComCapabilities_t *cap );
```


tmol*Com*Open

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

tmol*Com*Close

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

tmol*Com*GetInstanceSetup

```
extern tmLibappErr_t tmolComGetInstanceSetup(
Int instance, ptmolComInstanceSetup_t *setup );
```

tmol*Com*InstanceSetup

```
extern tmLibappErr_t tmolComInstanceSetup(
Int instance, ptmolComInstanceSetup_t setup );
```

tmol*Com*InstanceConfig

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

tmol*Com*Start

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

tmol*Com*Stop

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

- A type definition of the OL instance variable structure must not present in tmol-*Com*.h. It must be in a source file or private header file of the component.

tmal*Com*.h

- All input and output IDs of the components must be defined as macros, where the component name is the initial part of the macro name and is represented here by "*COM*":

```
#define COM_main_input  0
#define COM_main_output 0
...
```

- All component-specific error codes are defined as macros based on the component error base. The component name is the initial part of the macro name and is represented here by "*COM*." The macro name for the component error base incorporates the component name, represented here by "*Com*." See tmLibappErr.h on page 115.

```
#define COM_ERR_err1   ( Err_base_Com + 0x0001 )
#define COM_ERR_err2   ( Err_base_Com + 0x0002 )
...
```

- In the type definition of the structure **tmal*Com*Capabilities_t**, the first member must be a pointer to a structure of type **tsaDefaultCapabilities_t**:

```
typedef struct {
   ptsaDefaultCapabilities_t defaultCapabilities;
   ...
} tmalComCapabilities_t, ptmalComCapabilities_t;
```

- In the type definition of the structure **tmalComInstanceSetup_t**, the first element must be a pointer to a structure of type **tsaDefaultInstanceSetup_t**:

```
typedef struct {
   ptsaDefaultInstanceSetup_t defaultSetup;
   ...
} tmalComInstanceSetup_t, ptmalComInstanceSetup_t;
```

- Component configuration commands must be enumerated. In the enumeration, *"COM"* and *"Com"* represent the component name:

```
typedef enum {
   COM_COMMAND1 = tsaCmdUserBase,
   COM_COMMAND2 = tsaCmdUserBase + 0x01,
   ...
} tmalComCommands_t;
```

tsaCmdUserBase is defined in tsa.h and its value is currently 0x50. See also **tsaDefaultControlMessage_t** on page 190.

- Eight basic TSSA AL functions are declared with appropriate arguments

tmal*Com*GetCapabilities

```
extern tmLibappErr_t tmalComGetCapabilities(ptmalComCapabilities_t *cap);
```

tmal*Com*Open

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

tmal*Com*Close

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

tmal*Com*GetInstanceSetup

```
extern tmLibappErr_t tmalComGetInstanceSetup( Int instance,
ptmalComInstanceSetup_t *setup );
```

tmal*Com*InstanceSetup

```
extern tmLibappErr_t tmalComInstanceSetup( Int instance,
ptmalComInstanceSetup_t setup );
```

tmal*Com*InstanceConfig

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

tmal*Com*Start

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


tmal*Com*Stop

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

- At least one AL data process function is declared with instance as first argument.
- Type definition of the AL instance variable structure is not present. It must be in a source file or private header file of the component.

tmLibappErr.h

- **Err_base_***Com* must be defined after consulting tmLibappErr.h to avoid having the same error codes as other existing TriMedia components.

  The component *type* must be one of the following:

  ```
  GENERIC   SYSTEM   GRAPHICS   VIDEO   AUDIO   COMM   OTHER.
  ```

  ```
  #define Err_base_Com ( ERR_LAYER_TMAL | ERR_TYPE_type | 0x0100000 )
  ```

  Again, *"Com"* represents the name of the component.

Library Code

Every TSSA component must have a tmol*Com*.c and a tmal*Com*.c as part of the component library. Both tmol*Com*.c and tmal*Com*.c must at least adhere to the following rules for the component to be considered TSSA-compliant. For more example code, see *CopyIO Example and Explanation* in Chapter 10.

tmol*Com*.c

- **DEFAULT_STACK_SIZE** must be defined for task-based components.

  ```
  #define DEFAULT_STACK_SIZE 10000
  ```

  To calculate the appropriate stack size for a task-based component, use the pSOS function, **t_taskinfo**. For more details, see *Stack Calculation* in Chapter 8 of Book 4, *Software Tools*, Part A.

- You must define **TEXT_MEMORY_REQUIREMENT**, **DATA_MEMORY_REQUIREMENT**, and **PROCESSOR_REQUIREMENT** and give them values after component development. To calculate memory requirements, see *Calculating Memory Requirements* on page 104.
- You must define **INST_OPEN_MASK**, **INST_OPEN_MAGIC**, **INST_SETUP_MASK**, and **INST_SETUP_MAGIC**. **INST_OPEN_MAGIC** and **INST_SETUP_MAGIC** must be defined with "reasonably random component code" to identify the component.

  ```
  #define INST_OPEN_MASK    0x0000FFFF
  #define INST_OPEN_MAGIC   0x00001234
  #define INST_SETUP_MASK   0xFFFF0000
  #define INST_SETUP_MAGIC  0x12340000
  ```

- The type definition of the OL instance variable structure can be placed here or in an internal component header file. Use a name that includes "**ol***Com*" to prevent confusion between OL and AL instance variables.

  Its first member must be a pointer to a structure of type **tsaDefaultInstVar_t**.

  The structure includes a pointer to the OL instance setup structure of type **tmolComInstanceSetup_t** to be returned by **tmolComGetInstanceSetup**.

```
typedef struct {
   ptsaDefaultInstVar_t    defInstVars;
   ptmolComInstanceSetup_t setup;
   ...
} InstVars_t, *pInstVars_t;
```

- The use of global variables is strongly discouraged.
  - All internal variables for the OL layer should be in the instance variables structure.
  - The instance variables structure should be **malloc'd**, making its value unique.
  - To avoid name space pollution, any other global variables that cannot be included in the instance variables structure must be prefixed with **_tmol***Com*.
- All functions for the OL layer must be declared as one of the following:
  - **extern** for external API use.
  - **static** for internal use within its own file.
  - (To avoid name space pollution) all internal functions that cannot be declared static because they are used in more than one file in the OL layer must be prefixed with **_tmol***Com*, where *Com* represents the component name.
- The AL function table of type **tsaDefaultFuncs_t** must be statically declared and filled with appropriate functions.
- Eight basic TSSA OL functions exist.
  - Each calls a corresponding default function with appropriate arguments.
  - Using TSSA-provided macros, **tsaCheckOpen** and **tsaCheckSetup**, each checks whether the instance is opened and/or setup.
- **tmol***Com***Open**
  - Allocates OL instance variable structure and all its elements and gives them default values.
  - Sets default instance setup pointer in **setup** in the instance variables structure to that allocated in **tsaDefaultOpen**.
  - Sets **stackSize** to **DEFAULT_STACK_SIZE** for task-based components.
  - Returns a pointer to the OL allocated instance variable structure.
- **tmol***Com***Close**
  - Deallocates all memory allocated in **tmol***Com***Open**.

- **tmol*Com*GetInstanceSetup**
  - — Returns **setup** pointer from instance variables structure.
  - — Must not allocate a new **tmolInstanceSetup** structure each time.
  - — Must work regardless of whether the component is running.
- **tmol*Com*InstanceSetup**
  - — masks default magic with **INST_SETUP_MAGIC**.
- **tmol*Com*Start**
  - — Launches a thread (a task or an ISR) as result of calling **tsaDefaultStart**.
- **tmol*Com*Stop**
  - — Causes the thread to exit its processing loop.
- Repeated calls to **tmol*Com*Start** and **tmol*Com*Stop** works in arbitrary and repeated order. See *Example/Test Code* below.

tmal*Com*.c

- You must define **TEXT_MEMORY_REQUIREMENT**, **DATA_MEMORY_REQUIREMENT**, and **PROCESSOR_REQUIREMENT** and give them values after component development. To calculate memory requirements, see *Calculating Memory Requirements* on page 104.
- **NUM_SUPPORTED_INSTANCES** must be defined (–1 indicates unlimited instances supported).
- You must define **COM_OPEN_MASK**, **COM_OPEN_MAGIC**, **COM_SETUP_MASK**, **COM_SETUP_MAGIC**. **COM_OPEN_MAGIC** and **COM_SETUP_MAGIC** must be defined with "reasonably random component code" to identify component. *"COM"* represents the name of the component.

```
#define COM_OPEN_MASK    0x0000FFFF
#define COM_OPEN_MAGIC   0x00004321
#define COM_SETUP_MASK   0xFFFF0000
#define COM_SETUP_MAGIC  0x43210000
```

- The type definition of the AL instance variable structure can be placed in tmal*Com*.c or in an internal component header file. Use a name that includes "**al***Com*" to prevent confusion when looking at instance variables.

  The structure must include the parent of the AL layer, to be used for callback functions.

  The structure must include a member of type **tsaCompState_t**.

  The structure must include a pointer to the AL instance setup structure, of type **tmal*Com*InstanceSetup_t**, to be returned by **tmal*Com*GetInstanceSetup**.

```
typedef struct {
   UInt32                parent;
   tsaCompState_t        componentState;
```

```
    ptmalComInstanceSetup_t setup;
    ...
} InstVars_t, *pInstVars_t;
```

- All variables for the AL layer must conform to the following rules:
  - — All internal variables for the AL layer should be in the instance variables structure.
  - — The instance variables structure should be **malloc**'d, making its value unique.
  - — To avoid name space pollution, any global variables (that cannot be included in the instance variables structure) must be prefixed with **_tmal***Com*.
- All functions for the AL layer must be declared as one of the following:
  - — **extern** for external API use.
  - — **static** for internal use within its own file.
  - — (To avoid name space pollution) all internal functions that cannot be declared static because they are used in more than one file in the AL layer must be prefixed with **_tmal***Com*, where *Com* represents a component name.
- **COM_NUMBER_OF_INPUTS** and **COM_NUMBER_OF_OUTPUTS** must be defined, where *"COM"* represents the component name.

```
#define COM_NUMBER_OF_INPUTS  1
#define COM_NUMBER_OF_OUTPUTS 1
```

- Arrays of input and output formats must be declared in the default capabilities structure. Individual formats for each input or output can have data types and sub-types OR'd together to represent all the possible formats handled.
- The default capabilities structure, of type **tsaDefaultCapabilities_t**, must be statically declared and its members given appropriate values.
- The component capabilities structure, of type **tmal***Com***Capabilities_t**, must be statically declared and its members given default values from the default capabilities structure.
- Eight basic TSSA OL functions exist.
- **tmal***Com***GetCapabilities**

  Returns the component's capabilities structure.
- **tmal***Com***Open**
  - — Allocates an AL instance variables structure and all its elements and gives them default values.
  - — Masks default "magic" with **COM_OPEN_MAGIC**. You can use the component error base as a magic number.
  - — Allocates an instance setup structure to be returned by **tmal***Com***GetInstanceSetup**.
  - — Returns a pointer to the AL-allocated instance variables structure.
- **tmal***Com***Close**
  - — Deallocates all memory allocated by **tmal***Com***Open**.

- **tmal*Com*GetInstanceSetup**
  - — Returns the **setup** pointer from instance variables structure.
  - — Must not allocate a new **tmalInstanceSetup** structure each time.
- **tmal*Com*InstanceSetup**
  - — Gives values to instance variables structure members obtained from the instance setup structure.
  - — Masks default "magic" with **COM_OPEN_MAGIC**.
  - — Checks if any needed parameters are missing.
- **tmal*Com*Start**
  - — Uses default datain and dataout callback functions.
  - — Does not allocate or free memory if possible; all required memory should be allocated before streaming starts.
  - — Uses macros for input and output IDs defined in tmalCom.h.
  - — Install formats on output queues: Most components derive from the incoming packets the format of the outgoing packets. Hence components should install the format on their outputs by calling the progress function with flags tsaProgressFlagChangeFormat and description equal to the new format.
  - — Do not check for incoming null packets. Check return value of datain function. Packet cannot be Null without error. Handle format change error correctly.
  - — Do you handle format change messages? Task-based components should handle the **TMLIBAPP_NEW_FORMAT** error value in response to the datain function. Alternatively, you might not handle this message, but instead fill in the receiverFormatSetup member of the default capabilities function. That receiverFormatSetup alternative is provided so that ISR-based components can install a new format (and do the associated setup) from a context other than the ISR. Handling the **TMLIBAPP_NEW_FORMAT** error is the preferred implementation, when possible. Do one or the other, but not both.
  - — Be sure that your AL layer start function notifies the rest of the application if it exits the start loop with an error. The error callback function is a convenient way to handle this.
- **tmal*Com*Stop**

  Returns all packets to queues.
- All memory allocation done after **tmal*Com*Open** and **tmal*Com*InstanceSetup** must use the memory allocation callback function.
- Board-specific aspects of components must go through the BSP/registry.

  If a component uses board-specific hardware, it must query the BSP/registry to find out whether a function is supported and not access the hardware device directly.

Documentation

In addition to the API documentation that is expected, TSSA component documentation should discuss the capabilities and uses of the various inputs and outputs. It should discuss the use of the progress and error functions. Documentation should state explictly whether the progress and error functions are re-entrant.

Example/Test Code

- Both OL and AL layer examples exist.

  Filter components

  — Must be connected to digitizer and renderer components.

  — Examples should be named exolFilter and exalFilter.

  Digitizer and renderer components

  — Must be connected to corresponding digitizer or renderer components.

  — Examples should be named exol*type*IO and exal*type*IO, where *type* is the type of communication, e.g., exolFileIO.

  System examples

  — Examples that represent entire systems should be named ex*type*Sys, where *type* is the system demonstrated, e.g., exAudSys.

- OL layer must pass TSSA stop test.

  Filter components

  — Can be tested with digitizer and renderer components.

  — Can be tested with Fread and Fwrite with input files.

  Digitizer and renderer components

  — Can be tested with corresponding digitizer or renderer components.

  — Renderer components can be tested with Fread with input files.

  — Digitizer components can be tested with Fwrite.

- Examples and test code use macros for input and output IDs defined in tmal*Com*.h.

Chapter 13

tsa.h: Software Architecture Definitions

Default Capabilities Structure

This section presents the default capabilities structure found in the file tsa.h.

tsaDefaultCapabilities_t

```
typedef struct tsaDefaultCapabilities {
   tmComponentClass_t   componentClass;
   tmVersion_t          version;
   UInt32               capabilityFlags;
   Int                  textmemoryRequirement;
   Int                  datamemoryRequirement;
   Int                  processorRequirement;
   UInt                 numSupportedInstances;
   UInt                 numCurrentInstances;
   UInt                 numberOfInputs;
   ptmAvFormat_t       *inputFormats;
   UInt                 numberOfOutputs;
   ptmAvFormat_t       *outputFormats;
   tsaReceiverFormatSetupFunc_t  receiverFormatSetup;
} tsaDefaultCapabilities_t, *ptsaDefaultCapabilities_t;
```

Fields

`componentClass`	Component class indication, see tmAvFormats.h.
`version`	Component version number.
`capabilityFlags`	Indicates the presence or absence of optional features. See **tsaCapabilityFlags_t** on page 125.
`textmemoryRequirement`	Indication of the instruction memory requirements when the component is loaded.
`datamemoryRequirement`	Indication of the data memory requirements when the component is loaded.
`processorRequirement`	Indication of the processor load when the component is running.
`numSupportedInstances`	Maximum number of instances of this component that can be created. –1 means the number is limited only by system resources.
`numCurrentInstances`	Number of instances of this component that presently exist.
`numberOfInputs`	Number of inputs supported by this component.
`inputFormats`	Array of format structures that describe the formats accepted by each of the inputs supported by the component.
`numberOfOutputs`	Number of outputs supported by this component.
`outputFormats`	Array of format structures that describe the formats accepted by each of the outputs supported by the component.

Description

The default capabilities structure gives a description of a component that must be supported by every TSA component. Each component has a component capability structure that must include a pointer to a default capabilities struct as its first element. The formats in *inputFormats* and *outputFormats* in the capabilities structure are used to validate connections between any pair of components.

tsaCapabilityFlags_t

```
typedef enum {
   tsaCapFlagsNone                    = 0x00000000,
   tsaCapFlagsSupportsFormatReconfig  = 0x00000001,
   tsaCapFlagsSupportsControlQueue    = 0x00000002,
   tsaCapFlagsCopybackDatain          = 0x00000004,
   tsaCapFlagsInvalidateDataout       = 0x00000008,
   tsaCapFlagsInPlace                 = 0x00000010,
} tsaCapabilityFlags_t;
```

Fields

`tsaCapFlagsNone`	Default (no flags).
`tsaCapFlagsSupportsFormatReconfig`	This flag is set if the component reads the format field of every incoming packet and can dynamically switch formats based on this information.
`tsaCapFlagsSupportsControlQueue`	This flag is set if the component uses a queue-based control interface. In order to use it, the application must provide command and response queues. See **tsaDefaultControlDescriptorCreate** on page 143.
`tsaCapFlagsCopybackDatain`	This flag is set if the component requires a copy-back operation before another component can access the packet data.
`tsaCapFlagsInvalidateDataout`	This flag is set if the component requires an invalidate operation before another component can access the packet data.
`tsaCapFlagsInPlace`	This flag is set if the component sends data packets from the input full queue directly to the output full queue and does access any empty queues. The input and output components of the asynchronous feedback component share one empty queue while it is running. Queues are rearranged in the defaults during start and stop and is transparent to the user.

Default Instance Setup Structure

This section presents the default instance setup structure found in the file tsa.h.

tsaDefaultInstanceSetup_t

```
typedef struct tsaDefaultInstanceSetup {
   Int                     qualityLevel;
   tsaErrorFunc_t          errorFunc;
   UInt32                  progressReportFlags;
   tsaProgressFunc_t       progressFunc;
   tsaCompletionFunc_t     completionFunc;
   tsaDatainFunc_t         datainFunc;
   tsaDataoutFunc_t        dataoutFunc;
   tsaMemallocFunc_t       memallocFunc;
   tsaMemfreeFunc_t        memfreeFunc;
   ptsaClockHandle_t       clockHandle;
   ptsaInOutDescriptor_t   *inputDescriptors;
   ptsaInOutDescriptor_t   *outputDescriptors;
   UInt32                  parentId;
   tsaControlFunc_t        controlFunc;
   tsaControlDescriptor_t  controlDescriptors;
   UInt32                  priority;
   char                    taskName[16];
   UInt32                  stackSize;
   UInt32                  taskFlags;
   Bool                    createNoTask;
   UInt32                  taskStartArgument;
   UInt32                  reserved;
} tsaDefaultInstanceSetup_t, *ptsaDefaultInstanceSetup_t;
```

Fields

`qualityLevel`	Allow users to specify the quality of service of the component. Higher quality indicates a higher processor load.
`errorFunc`	Pointer to an error function. See **tsaErrorFunc_t** on page 167.
`progressReportFlags`	Set of flags that indicate when the component instance is required to report its progress. The interpretation of these flags will be component-specific.
`progressFunc`	Pointer to a progress function. See **tsaProgressFunc_t** on page 170.
`completionFunc`	Pointer to a completion function. See **tsaCompletionFunc_t** on page 173.

`datainFunc`	Pointer to a data input function. See **tsaDatainFunc_t** on page 177.
`dataoutFunc`	Pointer to a data output function. See **tsaDataoutFunc_t** on page 181.
`memallocFunc`	Pointer to a memory allocation function. See **tsaMemallocFunc_t** on page 184.
`memfreeFunc`	Pointer to a memory free function. See **tsaMemfreeFunc_t** on page 186.
`clockHandle`	Pointer to the clock that this component instance can use to interpret time stamps it receives in data packets. See **tsaClockHandle_t** on page 129.
`inputDescriptors`	Pointer to an array of input descriptor pointers. See **tsaInOutDescriptor_t** on page 130.
`outputDescriptors`	Pointer to an array of output descriptor pointers. See **tsaInOutDescriptor_t** on page 130.
`parentId`	Parent (creator) instance. Used as an parameter to callbacks. In the setup of AL layer components, this is the default instance variable ID.
`controlFunc`	Pointer to a control function. See **tsaControlFunc_t** on page 188.
`controlDescriptors`	Contains command and response queues. See **tsaControlDescriptor_t** on page 135.
`priority`	Operating system priority level required for this component.
`taskName`	The name of the task is used in debugging. pSOS uses the first four letters of this name. A default name is assigned when a component is opened, but it can be overridden by the user.
`stackSize`	Amount of stack memory to be used. A default stack size is assigned when a component is opened.
`taskFlags`	The operating system (pSOS) allows some specification of a task's properties (for example, whether it is preemptible or is time sliced). A default value is assigned when a component is opened, which can be overridden by the user.
`createNoTask`	Should be set in the OL layer of components that do not use a task before calling tsaDefaultInstanceSetup.
`taskStartArgument`	Should be set in the OL layer of components that want the task start argument of the component task to be something other than the default, before calling tsaDefaultStart. The default is the AL instance id.

`reserved`	Reserved.

Description

The instance setup structure contains information needed to set up and start a component. Because much of this information is common to most components, a default instance setup structure is defined. The component instance setup structure of each TSSA component has a pointer to a default instance setup structure as the first element.

Note:
All task attributes (priority, name, stack size) are part of the default instance setup (disp) save one. The taskStatus is part of the default instance variable (divp). The default instance variable contains a pointer to the default instance setup (divp->disp).

Clock Handle

tsaClockHandle_t

```
typedef struct tsaClockHandle_t {
   Int   clock;
} tsaClockHandle_t, *ptsaClockHandle_t;
```

Fields

`clock`	tsaClock instance ID.

Description

The clock handle gives a TSA component access to a time reference. Data packets received or sent by the component may have a time stamp associated with them. The clock handle can be used to interpret the time stamp.

InOutDescriptors

This section presents the InOutDescriptor structure found in the file tsa.h.

tsaInOutDescriptor_t

```
typedef struct tsaInOutDescriptor {
   ptmAvFormat_t              format;
   tsaInOutDescSetupFlags_t   flags;
   Bool                       receiverStopped;
   Bool                       cmdFullWakeupSent;
   Bool                       cmdEmptyWakeupSent;
   ptsaDefaultCapabilities_t  senderCap;
   ptsaDefaultCapabilities_t  receiverCap;
   UInt32                     senderIndex;
   UInt32                     receiverIndex;
   UInt32                     fullQueue;
   UInt32                     emptyQueue;
   ptmAvPacket_t              packetArray;
   ptmAvHeader_t              headerArray;
   UInt8                     *dataArray;
   UInt32                     packetBase;
   UInt32                     numberOfPackets;
   tsaIODescState_t           senderState;
   tsaIODescState_t           receiverState;
   ptmAvFormat_t              lastFormat;
   UInt32                     waitSemaphore;
   UInt32                     reserved;
} tsaInOutDescriptor_t, *ptsaInOutDescriptor_t;
```

Fields

`format`	The formats supported by this input or output. See tmAvFormats.h.
`flags`	Flags for the creation of tsaInOutDescriptor. See **tsaDefaultInOutDescriptorCreate** on page 141.
`receiverStopped`	Used by the default functions. Reserved.
`cmdFullWakeupSent`	Used by the default functions. Reserved.
`cmdEmptyWakeupSent`	Used by the default functions. Reserved.
`senderCap`	Pointer to sender's default capability structure. Used to check for compatibility.
`receiverCap`	Pointer to receiver's default capability structure. Used to check for compatibility.
`senderIndex`	The ID of the sender's output port.
`receiverIndex`	The ID of the receiver's input port.

`fullQueue`	The queue ID used for full packets.
`emptyQueue`	The queue ID used for empty packets.
`packetArray`	Pointer to the memory allocated to packets in this **tsaInOutDescriptor_t**. The allocation is done by **tsaDefaultInOutDescriptorCreate**.
`headerArray`	Pointer to the memory allocated to headers in this **tsaInOutDescriptor_t**. The allocation is done by **tsaDefaultInOutDescriptorCreate**.
`dataArray`	Pointer to the memory allocated to data buffers in this tsaInOutDescriptor_t. The allocation is done by **tsaDefaultInOutDescriptorCreate**.
`packetBase`	The base ID number given to this series of packets at creation.
`numberOfPackets`	The number of packets circulating in this **tsaInOutDescriptor_t**.
`senderState`	Used by the default functions. Reserved.
`receiverState`	Used by the default functions. Reserved.
`lastFormat`	Used by the default functions. Reserved.
`waitSemaphore`	When a component is setup to block on multiple inputs, the blocking semaphore is stored here.
`reserved`	Reserved.

Description

An **InOutDescriptor** describes the connection between two components. It is created and initialized by the **tsaDefaultInOutDescriptorCreate** function, using information from an argument of the type **tsaInOutDescriptorSetup_t**.

tsaInOutDescriptorSetup_t

```
typedef struct tsaInOutDescriptor {
   ptmAvFormat_t              format;
   tsaInOutDescSetupFlags_t   flags;
   String                     fullQName;
   String                     emptyQName;
   UInt32                     queueFlags;
   ptsaDefaultCapabilities_t  senderCap;
   ptsaDefaultCapabilities_t  receiverCap;
   UInt32                     senderIndex;
   UInt32                     receiverIndex;
   UInt32                     packetBase;
   UInt32                     numberOfPackets;
   UInt32                     numberOfBuffers;
   UInt32                     bufSize[1];
} tsaInOutDescriptorSetup_t, *ptsaInOutDescriptorSetup_t;
```

Fields

`format`	The formats supported by this input or output. See tmAvFormats.h.
`flags`	Flags for the creation of tsaInOutDescriptor. See **tsaInOutDescSetupFlags_t**.
`fullQName`	Name of the full queue to be created. Used for debugging.
`emptyQName`	Name of the empty queue to be created. Used for debugging.
`queueFlags`	Describes the properties of the queues to be created. The usual value given is **tmosQueueFlagsStandard**. See tmos.h.
`senderCap`	Pointer to sender's default capability structure. Used to check for compatibility.
`receiverCap`	Pointer to receiver's default capability structure. Used to check for compatibility.
`senderIndex`	The ID of the sender's output port.
`receiverIndex`	The ID of the receiver's input port.
`packetBase`	The base ID number given to this series of packets at creation.
`numberOfPackets`	The number of packets circulating in this **tsaInOutDescriptor_t**.
`numberOfBuffers`	The number of data buffers in each packet to be created.
`bufSize`	An array of integers giving the number of bytes to be allocated for each buffer. As a default, one

buffer is allocated. The allocation of packets with more than one buffer is described in Chapter 7, *TSSA Essentials*.

Description

The **tsaInOutDescriptorSetup_t** structure is filled in by the application and passed to the **tsaDefaultInOutDescriptorCreate** function, for creating a **tsaInOutDescriptor_t**. See **tsaDefaultInOutDescriptorCreate** on page 141.

tsaInOutDescSetupFlags_t

```
typedef enum {
   tsaIODescSetupFlagNone               = 0x00000000,
   tsaIODescSetupFlagCacheMalloc        = 0x00000001,
   tsaIODescSetupFlagMultiProc          = 0x00000002,
   tsaIODescSetupFlagCopybackDatain     = 0x00000004,
   tsaIODescSetupFlagInvalidateDataout  = 0x00000008,
} tsaIODescSetupFlags_t;
```

Fields

`tsaIODescSetupFlagNone` — Default flag.

`tsaIODescSetupFlagCacheMalloc` — This flag is set if the user wants the packets in this **tsaInOutDescriptor_t** to be created using **_cache_malloc** and destroyed using **_cache_free**.

`tsaIODescSetupFlagMultiProc` — This flag is set if this **tsaInOutDescriptor_t** is used between components on two processors.

`tsaIODescSetupFlagCopybackDatain` — This flag is set if the user wants the packets in this **tsaInOutDescriptor_t** to be copied back after creation.

`tsaIODescSetupFlagInvalidateDataout` — This flag is set if the user wants the packets in this **tsaInOutDescriptor_t** to be invalidated after creation.

ControlDescriptors

tsaControlDescriptor_t

```
typedef struct tsaControlDescriptor {
   UInt32                     commandQueue;
   UInt32                     responseQueue;
   tsaControlDescSetupFlags_t flags;
   UInt32                     reserved;
} tsaControlDescriptor_t, *ptsaControlDescriptor_t;
```

Fields

`commandQueue`	Used to send commands from the application to the component.
`responseQueue`	Used to send responses from the component to the application.
`flags`	Flags for the creation of **tsaControlDescriptor_t**. See **tsaControlDescSetupFlags_t**.
`reserved`	Reserved.

Description

A **tsaControlDescriptor_t** is used by the application to send commands to a component. It is created and initialized by the **tsaDefaultControlDescriptorCreate** function, using information from an argument of the type **tsaControlDescriptorSetup_t**. It is only used if the **tsaCapFlagsSupportsControlQueue** is set in the component's capability flags.

tsaControlDescriptorSetup_t

```
typedef struct tsaControlDescriptor {
   UInt32                        commandQName;
   UInt32                        responseQName;
   UInt32                        queueFlags;
   tsaControlDescSetupFlags_t    flags;
} tsaControlDescriptor_t, *ptsaControlDescriptor_t;
```

Fields

`commandQName`	Name of the command queue to be created. Used for debugging.
`responseQuame`	Name of the response queue to be created. Used for debugging.
`queueFlags`	Describes the properties of the queues to be created. The usual value given is **tmosQueueFlagsStandard**. See tmos.h.
`flags`	Flags for the creation of **tsaControlDescriptor_t**. See page 135.

Description

The **tsaControlDescriptorSetup_t** structure is filled in by the application and passed to the **tsaDefaultControlDescriptorCreate** function, for creating a **tsaControlDescriptor_t**. See **tsaDefaultControlDescriptorCreate** on page 143.

tsaControlDescSetupFlags_t

```
typedef enum {
   tsaControlDescSetupFlagNone = 0x00000000,
} tsaControlDescSetupFlags_t;
```

Fields

`tsaControlDescSetupFlagNone`	Currently, there are no flags required for the creation of a **tsaControlDescriptor_t**.

Default Instance Variables

This section presents the default instance variable structure found in the file tsa.h.

tsaDefaultInstVar_t

```
typedef struct tsaDefaultInstVar {
   UInt32                      magic;
   ptsaDefaultInstanceSetup_t  instSetup;
   Int                         tmalInstance;
   ptsaDefaultFuncs_t          tmalFunc;
   volatile tsaTaskStatus_t    taskstatus;
   UInt32                      task;
   UInt32                      numberOfInputs;
   UInt32                      numberOfOutputs;
   UInt32                      stopSemaphore;
   UInt32                      configSemaphore;
   Int                         periodOfComponent;
   UInt32                     *outputEmptyQueues;
   UInt32                      reserved;
} tsaDefaultInstVar_t, *ptsaDefaultInstVar_t;
```

Fields

`magic`	Magic number used to identify the validity of an instance ID, by checking whether is it open and/or setup. It is set by the OL layer of a component.
`instSetup`	Pointer to default setup structure of the instance.
`tmalInstance`	The associated tmal instance ID.
`tmalFunc`	AL layer default API function pointers See **tsaDefaultFuncs_t** on page 139.
`taskstatus`	The current task status.
`task`	The associated task ID, if it exists.
`numberOfInputs`	The number of input connections (**tsaInOutDescriptor_t**) this instance can have.
`numberOfOutputs`	The number of output connections (**tsaInOutDescriptor_t**) this instance can have.
`stopSemaphore`	Used to ensure mutual exclusion in the access of the component task. Also used to communicate the completion of the stop sequence between the component and the application calling the default stop function. Its usage is transparent to the application.
`configSemaphore`	Used to ensure the completion of a sequence of command and response by the application and

	component, respectively. Its usage is transparent to the application.
periodOfComponent	This parameter should represent the average time during one loop iteration in a task. It can be used to tune the behavior of the scheduling algorithm that comes into play when a task is connected to an interrupt-based consumer of data.
outputEmptyQueues	Used to store the output empty queues of an in-place component, while it is running. See **tsaCapabilityFlags_t** on page 125.
reserved	Reserved.

Description

The default instance variable is used to identify common elements of all TSSA components. It is used by the default functions.

Default AL Function Table

tsaDefaultFuncs_t

```
typedef struct tsaDefaultFuncs {
   tsaGetCapabilitiesFunc_t    getcapabilitiesFunc;
   tsaOpenFunc_t               openFunc;
   tsaCloseFunc_t              closeFunc;
   tsaInstanceSetupFunc_t      instancesetupFunc;
   tsaStartFunc_t              startFunc;
   tsaStopFunc_t               stopFunc;
   tsaInstanceConfigFunc_t     instanceconfigFunc;
   tsaGetCapabilitiesFuncM_t   getcapabilitiesFuncM;
   tsaOpenFuncM_t              openFuncM;
} tsaDefaultFuncs_t, *ptsaDefaultFuncs_t;
```

Fields

`getcapabilitiesFunc`	The AL layer GetCapabilites function of the component.
`openFunc`	The AL layer Open function of the component.
`closeFunc`	The AL layer Close function of the component.
`instancesetupFunc`	The AL layer InstanceSetup function of the component.
`startFunc`	The AL layer Start function of the component.
`stopFunc`	The AL layer Stop function of the component.
`instanceconfigFunc`	The AL layer InstanceConfig function of the component.
`getcapabilitiesFuncM`	The AL layer GetCapabilitiesM function of the component.
`openFuncM`	The AL layer OpenM function of the component.

Description

The **tsaDefaultFuncs_t** table is a list of standard AL functions of a component. It is used by the default functions to access the API of the associated AL layer of the component. getcapabilitiesFuncM and openFuncM are used by components that can access multiple units of a hardware device.

Default Utility Functions

This section describes the utility functions that are defined in the default TSSA library. These functions are meant to be used in TSSA applications..

tsaDefaultInOutDescriptorCreate

```
extern tmLibappErr_t tsaDefaultInOutDescriptorCreate (
   ptsaInOutDescriptor_t       *piodesc,
   ptsaInOutDescriptorSetup_t   psetup
);
```

Parameters

`piodesc`	Pointer to the **tsaInOutDescriptor_t** to be created.
`psetup`	Pointer to the **tsaInOutDescriptorSetup_t** structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NULL_IODESC`	Either piodesc or psetup is Null.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Not enough memory.
`TMLIBAPP_ERR_IN_PLACE`	Connection of two in-place components is not yet supported.
`TMLIBAPP_ERR_FORMAT_…`	Errors from the negotiation or installing of the format.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Allocates the memory for a **tsaInOutDescriptor_t**. If a format is specified, checks it against the capabilities of sender and receiver and install it if compatible. Creates the full and empty queues required for this connection. Allocates and initializes the packets for this **tsaInOutDescriptor_t**, and puts them on the empty queue. If the application does not want this function to create the packets, set **psetup–>numberOfPackets** to 0.

tsaDefaultInOutDescriptorDestroy

```
extern tmLibappErr_t tsaDefaultInOutDescriptorDestroy (
   ptsaInOutDescriptor_t   piodesc
);
```

Parameters

`piodesc`	Pointer to the **tsaInOutDescriptor_t** to be destroyed.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NULL_IODESC`	**piodesc** is Null.
`TMLIBAPP_ERR_NOT_STOPPED`	The sender or receiver component is not stopped.
`TMLIBAPP_ERR_FORMAT_…`	Errors from un-installing of the format.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Both sender and receiver components must be stopped. Releases the format. Destroys the queues after removing all packets from the empty queue. All packets should be in the empty queue when the components are stopped. Frees the memory associated with this **tsaInOutDescriptor_t**.

tsaDefaultControlDescriptorCreate

```
extern tmLibappErr_t tsaDefaultControlDescriptorCreate (
   ptsaControlDescriptor_t       *pcdesc,
   ptsaControlDescriptorSetup_t   csetup
);
```

Parameters

`pcdesc`	Pointer to the **tsaControlDescriptor_t** to be created.
`csetup`	Pointer to the **tsaControlDescriptorSetup_t** structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NULL_CTRLDESC`	Either **pcdesc** or **csetup** is Null.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Not enough memory.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Allocates the memory for a **tsaControlDescriptor_t**. Creates the command and response queues required for communication.

tsaDefaultControlDescriptorDestroy

```
extern tmLibappErr_t tsaDefaultControlDescriptorDestroy (
   ptsaControlDescriptor_t   pcdesc
);
```

Parameters

`pcdesc`	Pointer to **tsaControlDescriptor_t** to be destroyed.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NULL_CTRLDESC`	**pcdesc** is Null.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Destroys the queues after removing all packets from the both queues. Frees the memory associated with **tsaControlDescriptor_t**.

tsaDefaultSenderReconnect

```
extern tmLibappErr_t tsaDefaultSenderReconnect (
   ptsaInOutDescriptor_t      piodesc,
   UInt                       senderIndex,
   ptsaDefaultCapabilities_t  senderCap,
   ptmAvFormat_t              format
);
```

Parameters

`piodesc`	Pointer to **tsaInOutDescriptor_t** to which the sender is to be reconnected.
`senderIndex`	Output channel ID on new sender.
`senderCap`	Pointer to the new sender's default capabilities structure.
`format`	Format of new connection if different from existing format.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NULL_IODESC`	**piodesc** is Null.
`TMLIBAPP_ERR_NOT_STOPPED`	The current sender component is not stopped.
`TMLIBAPP_ERR_CAP_REQUIRED`	The new sender capability is required to reconnect the sender.
`TMLIBAPP_ERR_INVALID_CHANNEL_ID`	Invalid sender output ID.
`TMLIBAPP_ERR_FORMAT_…`	Errors from the negotiation or installing of the format.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Reconnects the **tsaInOutDescriptor_t** to another sender. If no format is specified, checks existing format against the capabilities of the sender. Otherwise, checks the specified format against the capabilities of sender and receiver and installs it if compatible.

tsaDefaultReceiverReconnect

```
extern tmLibappErr_t tsaDefaultReceiverReconnect (
   ptsaInOutDescriptor_t       piodesc,
   UInt                        receiverIndex,
   ptsaDefaultCapabilities_t   receiverCap
);
```

Parameters

`piodesc`	Pointer to **tsaInOutDescriptor_t** in which the receiver is to be reconnected.
`receiverIndex`	Input channel ID on new receiver.
`receiverCap`	Pointer to the new receiver's default capabilities structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NULL_IODESC`	**piodesc** is Null.
`TMLIBAPP_ERR_NOT_STOPPED`	The current receiver component is not stopped.
`TMLIBAPP_ERR_CAP_REQUIRED`	The new receiver capability is required to reconnect the receiver.
`TMLIBAPP_ERR_INVALID_CHANNEL_ID`	Invalid receiver input ID.
`TMLIBAPP_ERR_FORMAT_…`	Errors from the negotiation or installing of the format.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Reconnects the **tsaInOutDescriptor_t** to another receiver. Since the receiver cannot specify the format of a connection, checks existing format against the capabilities of the new receiver and installs it if compatible.

tsaDefaultInstallFormat

```
extern tmLibappErr_t tsaDefaultInstallFormat (
   ptsaInOutDescriptor_t   piodesc,
   ptmAvFormat_t           format
);
```

Parameters

`piodesc`	Pointer to **tsaInOutDescriptor_t** in which a new format is to be installed.
`format`	Pointer to the new format.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NULL_IODESC`	piodesc is Null.
`TMLIBAPP_ERR_FORMAT`	No new format specified.
`TMLIBAPP_ERR_FORMAT_…`	Errors from the negotiation or installing of the format.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Releases current format. Checks the specified format against the capabilities of sender and receiver. If compatible, creates a unique copy of the requested format, claims it, and installs it in the **tsaInOutDescriptor_t**. If necessary, calls the **receiverFormatSetup** function. See **tsaDefaultFuncs_t** on page 139.

Note: this function should not be called from within an interrupt service routine.

tsaDefaultUnInstallFormat

```
extern tmLibappErr_t tsaDefaultUnInstallFormat (
   ptsaInOutDescriptor_t   piodesc
);
```

Parameters

`piodesc`	Pointer to **tsaInOutDescriptor_t** in which the format is to be un-installed.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_NULL_IODESC`	**piodesc** is Null.
`TMLIBAPP_ERR_NOT_STOPPED`	The sender or receiver component is not stopped.
`TMLIBAPP_ERR_FORMAT_…`	Errors from un-installing of the format.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Releases current format in the **tsaInOutDescriptor_t**.

tsaDefaultSleep

```
extern tmLibappErr_t tsaDefaultSleep (
   Int   ticks
);
```

Parameters

`ticks`	Time to sleep is measured in operating system (pSOS) ticks (usually 10ms).

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

The sleep utility in a real-time operating system causes the current task to be scheduled out for the specified amount of time in ticks.

tsaDefaultCheckQueues

```
extern tmLibappErr_t tsaDefaultCheckQueues (
   ptsaInOutDescriptor_t   piodesc
);
```

Parameters

`piodesc`	Pointer to **tsaInOutDescriptor_t** to be checked

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_CHECK_QUEUES`	The number of packets found in the full and empty queues disagree with the original number of packets installed.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

To be used when components are stopped. Takes all packets out of the empty queue. Counts the number of data packets. Removes any non-data packets. DPs on each packet found. Puts all data packets back in the queue. Then, does the same for the full queue. Checks the number of packets found against the **piodesc->numberOfPackets**. DPs the results. All DPs in this function are set at level 1 (L1_DP).

Default API Functions

This section presents the default API functions found in the file tsa.h.

tsaDefaultGetCapabilities

```
extern tmLibappErr_t tsaDefaultGetCapabilities (
   Bool                *add_done,
   ptsaDefaultFuncs_t   tsafunc,
   UInt32              *cap,
   UInt32               text,
   UInt32               data,
   UInt32               proc
);
```

Parameters

`add_done`	The address of a variable that is false on the first invocation of this function, and true thereafter. When it is true, it means that the requirements of the OL and AL have been added together.
`tsaFunc`	Pointer to a default AL function table. This pointer is used to provide the AL GetCapabilities the functions to call.
`cap`	Pointer to a component specific capabilities structure to be returned from this function.
`text`	OL layer text memory requirement.
`data`	OL layer data memory requirement.
`proc`	OL layer processor requirement.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is Null.

Description

Designed to be called from tmolComGetCapabilities, the function calls the AL layer capabilities function and returns a pointer to the component specific capabilities structure in parameter, *cap*. The first time that it is called, it adds the memory requirements of the OL layer to those of the AL layer and sets **add_done** to true. The first call to the getCapabilities functions are likely to allocate memory that is never subsequently freed. For this reason, you may want to collect all of these calls in one place at the start of your program to minimize memory fragmentation.

tsaDefaultGetCapabilitiesM

```
extern tmLibappErr_t tsaDefaultGetCapabilitiesM (
   Bool                *add_done,
   ptsaDefaultFuncs_t   tsafunc,
   UInt32              *cap,
   UInt32               text,
   UInt32               data,
   UInt32               proc,
   unitSelect_t         unitNumber
);
```

Parameters

`add_done`	The address of a variable that is false on the first invocation of this function, and true thereafter. When it is true, it means that the requirements of the OL and AL have been added together.
`tsaFunc`	Pointer to a default AL function table. This pointer is used to provide the AL GetCapabilities functions to call.
`cap`	Pointer to a component specific capabilities structure to be returned from this function.
`text`	OL layer text memory requirement.
`data`	OL layer data memory requirement.
`proc`	OL layer processor requirement.
`unitNumber`	Unit number of a hardware device.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is Null.

Description

Designed to be called from tmolComGetCapabilitiesM(), the function calls the AL layer capabilities function and returns a pointer to the component specific capabilities structure in parameter, cap. The first time that it is called, it adds the memory requirements of the OL layer to those of the AL layer and sets **add_done** to true. **unitNumber** indicates the unit of the hardware device used by this component.The first call to the getCapabilities functions are likely to allocate memory that is never subsequently freed. For this reason, you may want to collect all of these calls in one place at the start of your program to minimize memory fragmentation.

tsaDefaultOpen

```
extern tmLibappErr_t tsaDefaultOpen (
   ptsaDefaultInstVar_t   *divp,
   UInt32                  magic,
   ptsaDefaultFuncs_t      tsafunc
);
```

Parameters

`divp`	Pointer to a default instance variable structure to be returned from this function.
`magic`	Magic string to determine the validity of `divp` during execution.
`tsaFunc`	Pointer to a default AL function table. This is installed into the **divp** for later use.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**divp** is not a valid default instance variable structure.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is Null.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Not enough memory.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Designed to be called from tmolComOpen(), the function allocates memory for the default instance variables. Then, it creates the **stopSemaphore** for mutual exclusion of task access. Finally, it calls the AL layer open function and returns a pointer to the default instance variables in the **divp** parameter.

tsaDefaultOpenM

```
extern tmLibappErr_t tsaDefaultOpenM (
   ptsaDefaultInstVar_t   *divp,
   UInt32                  magic,
   ptsaDefaultFuncs_t      tsafunc,
   unitSelect_t            unitNumber
);
```

Parameters

`divp`	Pointer to a default instance variable structure to be returned from this function.
`magic`	Magic string to determine the validity of **divp** during execution.
`tsaFunc`	Pointer to a default AL function table. It is installed into the divp for later use.
`unitNumber`	Unit number of a hardware device.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**divp** is not a valid default instance variable structure.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is Null.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Not enough memory.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Designed to be called from tmolComOpenM(), the function allocates memory for the default instance variables. Then, it creates the **stopSemaphore** (see **tsaDefaultInstVar_t**) for mutual exclusion of task access. Finally, it calls the AL layer open function and returns a pointer to the default instance variables in the **divp** parameter. **unitNumber** indicates the unit of the hardware device used by this component.

tsaDefaultClose

```
extern tmLibappErr_t tsaDefaultClose (
   ptsaDefaultInstVar_t   divp
);
```

Parameters

`divp`	Pointer to a default instance variable structure to be destroyed.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**divp** is not a valid default instance variable structure.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is Null.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Designed to be called from tmolComClose(), the function first calls the AL layer close function. Then it waits until the component is stopped before continuing. It then destroys the **stopSemaphore** (see **tsaDefaultInstVar_t**). Finally, it frees all memory associated with this **ptsaDefaultInstVar_t** structure that was allocated by the defaults.

tsaDefaultGetInstanceSetup

```
extern tmLibappErr_t tsaDefaultGetInstanceSetup (
   Int       instance,
   UInt32   **setup
);
```

Parameters

`instance`	ID of a default instance variable structure.
`setup`	Pointer to a pointer to a default instance setup structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_SETUP`	Setup is null or **divp->instSetup** is null.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is Null.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Calls the AL layer GetInstanceSetup function through the tmalFunc table. The address of the AL layer instance setup structure is returned. This function is designed to be used in the implementation of the required OL layer GetInstanceSetup function. The memory representing the instance setup structure should have been allocated during the preceeding call to tmalComOpen(), and it should be filled in with currently correct values, or defaults if the component has not yet been setup.

tsaDefaultInstanceSetup

```
extern tmLibappErr_t tsaDefaultInstanceSetup (
   Int       instance,
   UInt32   *setup
);
```

Parameters

`instance`	ID of a default instance variable structure.
`setup`	Pointer to a default instance setup structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Instance is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_SETUP`	Setup is null or **divp->instSetup** is null.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is Null.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Designed to be called from **tmol***Com***InstanceSetup**, this function passes down and saves (in the default instance variable, divp) all default callback functions to the AL layer and sets the AL parentID to instance. It then calls the AL instance setup function.

Note:
The passed instance value contains two pointers to the default instance setup structure. An assertion checks that these two pointers point to the same memory:

```
divp = *(ptsaDefaultInstVar_t *)instance;
dsp = *(ptsaDefaultInstanceSetup_t *)setup;
disp = divp->instSetup;
tmAssert(disp == dsp,TMLIBAPP_ERR_INVALID_SETUP);
```


tsaDefaultStart

```
extern tmLibappErr_t tsaDefaultStart (
   ptsaDefaultInstVar_t   divp
);
```

Parameters

`divp`	Pointer to a default instance variable structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**divp** is not a valid default instance variable structure.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is Null.
`TMLIBAPP_ERR_ALREADY_STARTED`	Instance is already started.
`TMLIBAPP_ERR_INVALID_SETUP`	In-place components must have the same number of inputs and outputs.
`TMLIBAPP_ERR_MISMATCHED_DESC`	In-place components must have the descriptors of a given channel attached.
`TMLIBAPP_ERR_FORMAT`	In-place components must have the formats of the input and output descriptors already installed.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Designed to be called from **tmol***Com***Start**, the function first acquires the lock (**stopSemaphore**) to access the instance. Note that when there is an error, **tsaDefaultStart** releases the lock and returns. Then, it sets the receiverState of all the inputDescriptor and the senderState of all the outputDescriptors to ACTIVE. If the component is an in-place component, it arranges the queues, by attaching the emptyQueue of the inputDescriptor directly to emptyQueue of the outputDescriptor and storing the output empty queues, thus bypassing this component. It sets the task status of the component to RUNNING. If the component is not task-based, it directly calls the AL start function. If the component is task-based, it creates and starts the default task, if it does not already exist. If it already exists, it resumes the current default task. The default task will then call the AL start function. Finally, **tsaDefaultStart** releases the lock and returns.

tsaDefaultStop

```
extern tmLibappErr_t tsaDefaultStop (
   ptsaDefaultInstVar_t   divp
);
```

Parameters

`divp`	Pointer to a default instance variable structure.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**divp** is not a valid default instance variable structure.
`TMLIBAPP_ERR_BAD_TMALFUNC_TABLE`	**tsaFunc** is null.
`TMLIBAPP_ERR_ALREADY_STOPPED`	Instance is already stopped.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system.

Description

Designed to be called from tmolComStop(), the function first acquires the lock (stopSemaphore) to access the instance. Note that when there is an error, **tsaDefaultStop** saves the first error and continues. Then, it calls the AL stop function, sets the task status to **STOP_REQUESTED**, for notifying subsequent Datain or Dataout functions calls from the instance.

If the component is task-based, it sends **WAKEUP** packets to queues that directed its task to break it out of waiting for a packet. It then waits for the default task to finish its stop sequence, on the **stopSemaphore**. At this point, the default task returns from the AL start function, sets the receiverState of all the inputDescriptor and the senderState of all the outputDescriptors to **STOPPED**, and flushes the input full and empty queues only. It now sets its task status to **NOT_STARTED**, if it was requested to stop. If the AL start function returned upon an error, the task status is left as **RUNNING** and the application must call **tsaDefaultStop** to get the instance back into a consistent state. When the task is done, **tsaDefaultStop** sets the task status to **NOT_STARTED**.

If the component is not task-based, **tsaDefaultStop** sets the receiverState of all the input descriptors and the senderState of all the output descriptors to **STOPPED**. It then sets the task status to **NOT_STARTED**.

If the component is an in-place component, it rearranges the queues back into its original configuration. Finally, **tsaDefaultStop** releases the lock and returns.

tsaDefaultInstanceConfig

```
extern tmLibappErr_t tsaDefaultInstanceConfig (
   ptsaDefaultInstVar_t   divp,
   UInt32                 flags,
   ptsaControlArgs_t      args
)
```

Parameters

`divp`	Pointer to a default instance variable structure.
`flags`	Use **tsaControlFlags_t** (page 189) or a cast to **UInt32**.
`args`	See **tsaControlArgs_t** on page 189.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**divp** is not a valid default instance variable structure.
`TMLIBAPP_ERR_INVALID_ARGS`	**args** is Null.
`TMLIBAPP_ERR_NULL_CONFIGFUNC`	AL instance config function is Null.
`TMLIBAPP_ERR_NO_QUEUE`	Control Descriptor not passed in at instance setup.
`TMLIBAPP_ERR_NOT_STARTED`	Instance is not started and therefore cannot respond to any commands.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system.

Description

Designed to be called from **tmol***Com***InstanceConfig**, the function configures an instance that uses control queue interface, while it is running. It uses the queues in the control descriptor passed to the instance during setup. Only one command can be processed at one time; i.e. the command/response sequence is atomic. **tsaDefaultInstanceConfig** first acquires the lock (**stopSemaphore**) for the instance. After a series of checks, it sends the command to the command queue. Then, it sends **WAKEUP** packets to data queues attached to the task. This will wake the component up when it was in a blocking wait for a packet. When the task wakes up it can check the command queue and the defaults will call the installed configuration function. Finally, **tsaDefaultInstanceConfig** waits for a response from the task on the response queue. It releases the lock on the instance and returns.

tsaDefaultStopPin

```
extern tmLibappErr_t tsaDefaultStopPin (
   Int             instance,
   tsaInOutPin_t   inOut,
   Int             pinId
);
```

Parameters

`instance`	A default instance variable structure.
inOut	Determines whether input or output pins.
`pinId`	Determines which pin.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instance** is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	**inOut** is not a valid **tsaInOutPin_t**.
`TMLIBAPP_ERR_CHANNEL_ID`	**pinId** is not a valid pin number for this **tsaInOutDescriptor_t**.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Designed to be called from **tmol***Com***StopPin**, the function allows the application to stop a specific pin of the component instance, while the instance continues to run. **inOut** indicates whether to stop the input or output pin and **pinId** indicates which channel. The pin state is set to **STOPPED**. When a pin is stopped, all packets on the full queue are flushed to the empty queue. Packets held by the components are not necessarily returned.

TIP

This function can be useful during development because it allows incremental integration. Write your application first and then start all components. Immediately after starting the component, call **tsaDefaultStopPin** for all input pins. When the input pin is stopped, all full packets are returned immediately to the empty queue. In this way, you can observe your source code in operation without processing the data.

tsaDefaultUnStopPin

```
extern tmLibappErr_t tsaDefaultUnStopPin (
   Int             instance,
   tsaInOutPin_t   inOut,
   Int             pinId
);
```

Parameters

`instance`	A default instance variable structure.
`inOut`	Determines whether input or output pins.
`pinId`	Determines which pin.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instance** is not a valid default instance variable structure. Cast to divp.
`TMLIBAPP_ERR_INVALID_ARGS`	**inOut** is not a valid **tsaInOutPin_t**.
`TMLIBAPP_ERR_CHANNEL_ID`	**pinId** is not a valid pin number for this **tsaInOutDescriptor_t**.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system (pSOS).

Description

Designed to be called from **tmol***Com***UnStopPin**, the function allows the application to unstop a previously stopped pin of the component. **inOut** indicates whether to unstop the input or output pin and **pinId** indicates which channel. The pin state is set to **ACTIVE**.

Default Callback Functions

This section presents the default callback functions, and associated the data structures and enumerated types contained in the file tsa.h.

tsaDefaultErrorFunction

```
extern tmLibappErr_t tsaDefaultErrorFunction (
   Int               instId,
   UInt32            flags,
   ptsaErrorArgs_t   args
);
```

Parameters

`instId`	Instance of a default instance variable structure. See **tsaDefaultInstVar_t** on page 137.
`flags`	Flags to be passed to application-specific error function, if exists.
`args`	Arguments to passed to application-specific error function, if exists.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instId** is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	**divp->instSetup** or **args** is null.

Description

If **instId** is null, the function DPs error code, Null instance, and **flags**. Otherwise, checks validity of **instId** (cast to **divp**) and **divp->setup**. Calls application-specific error function if exists and returns. All DPs in this function are set at level 1 (L1_DP).

tsaErrorFunc_t

```
typedef tmLibappErr_t (*tsaErrorFunc_t)(
   Int               instId,
   UInt32            flags,
   ptsaErrorArgs_t   args
);
```

Fields

`instId`	Parent (creator) of the component layer calling the function.
`flags`	Use error flags of type **tsaErrorFlags_t** or cast to **UInt32**.
`args`	Pointer to error function arguments of the type **tsaErrorArgs_t**.

Description

The error callback function is called by a component to report an error to its parent. **flags** is the error flag of type **tsaErrorFlags_t** or user-defined cast to **UInt32**. The type is **UInt32** in the function definition to allow user-defined flags without casting.

args is a pointer to a **tsaErrorArgs_t** structure, which contains the error code specific to the component. Component-specific error codes are described in the API documentation of the component.

tsaErrorFlags_t

```
typedef enum {
   tsaErrorFlagNone       = 0x00000000,
   tsaErrorFlagNonFatal   = 0x01000000,
   tsaErrorFlagFatal      = 0x02000000,
} tsaErrorFlags_t;
```

Description

This type is used by **tsaErrorFunc_t** functions. Indicates whether the error reported is fatal or non-fatal. See **tsaErrorFunc_t** on page 167.

tsaErrorArgs_t

```
typedef struct tsaErrorArgs {
   Int        errorCode;
   Pointer   description;
} tsaErrorArgs_t, *ptsaErrorArgs_t;
```

Description

This type is used by **tsaErrorFunc_t** functions. The **errorCode** identifies the error reported by a component and is enumerated in the component's header file. The **description** field is specific to the individual component. See **tsaErrorFunc_t** on page 167.

tsaDefaultProgressFunction

```
extern tmLibappErr_t tsaDefaultProgressFunction (
   Int                  instId,
   UInt32               flags,
   ptsaProgressArgs_t   args
);
```

Parameters

`instId`	Instance of a default instance variable structure. See **tsaDefaultInstVar_t** on page 137.
`flags`	Flags to be passed to application-specific error function, if exists.
`args`	Arguments to passed to application-specific error function, if exists.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instId** is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	**divp->instSetup** or args is null.

Description

If **instId** is Null, the function DPs progress code, Null instance, and **flags**. Otherwise, it checks the validity of **instId** (cast to **divp**) and **divp->setup**. If flags contains **tsaProgressFlagChangeFormat**, automatically installs new format and call the error callback function if it failed. Otherwise, calls application-specific progress function if exists and returns. Otherwise, DPs progress code, **instId**, instance name, and **flags**. All DPs (debug prints) in this function are set at level 1 (L1_DP).

Note: This function should not be called from within an interrupt service routine when the flags are set to **tsaProgressFlagChangeFormat**. That is, do not install a new format in an interrupt service routine.

tsaProgressFunc_t

```
typedef tmLibappErr_t (*tsaProgressFunc_t)(
   Int                  instId,
   UInt32               flags,
   ptsaProgressArgs_t   args
);
```

Fields

`instId`	Parent (creator) of the component layer calling the function.
`flags`	Use progress flags of type **tsaProgressFlags_t** or cast to **UInt32**.
`args`	Pointer to error function arguments of the type **tsaProgressFlags_t**.

Description

The progress callback function is called by a component to report any progress to its parent. **flags** is the progress flag of type **tsaProgressFlags_t** or cast to **UInt32**. The type is **UInt32** in the function definition to allow user-defined flags without casting.

args is a pointer to a **tsaProgressArgs_t** structure, which contains the progress code specific to the component. Component-specific progress codes are described in the API documentation of the component.

tsaProgressFlags_t

```
typedef enum {
   tsaProgressFlagNone           = 0x00000000,
   tsaProgressFlagEndOfStream    = 0x01000000,
   tsaProgressFlagChangeFormat   = 0x02000000,
} tsaProgressFlags_t;
```

Description

This type is used by **tsaProgressFunc_t** functions. It indicates progress codes resulting from the default layer. **tsaProgressFlagEndOfStream** synchronizes the stopping of a chain of components without losing any valid data. **tsaProgressFlagChangeFormat** is used when a sender component requests a format change in the **tsaInOutDescriptor_t**. See **tsaProgressFunc_t** on page 170. **tsaProgressFlagNone** is used as default flag.

tsaProgressArgs_t

```
typedef struct tsaProgressArgs {
   Int       progressCode;
   Pointer   description;
} tsaProgressArgs_t, *ptsaProgressArgs_t;
```

Description

This type is used by **tsaProgressFunc_t** functions. The **progressCode** identifies the progress reported by a component and is enumerated in the component's header file. The **description** field is specific to the individual component. See **tsaProgressFunc_t** on page 170.

tsaDefaultCompletionFunction

```
extern tmLibappErr_t tsaDefaultCompletionFunction (
   Int                    instId,
   UInt32                 flags,
   ptsaCompletionArgs_t   args
);
```

Parameters

`instId`	Instance of a default instance variable structure. See **tsaDefaultInstVar_t** on page 137.
`flags`	Flags to be passed to application-specific completion function, if it exists.
`args`	Arguments to passed to application-specific completion function, if it exists.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instId** is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	Either **divp->instSetup** or **args** is null.

Description

If **instId** is null, the function prints (through DP) the completion code, Null instance, and **flags**. Otherwise, it checks the validity of **instId** (cast to **divp**) and **divp–>setup**. It then calls the application-specific completion function if that exists and returns. All DPs in this function are set at level 1 (L1_DP).

tsaCompletionFunc_t

```
typedef tmLibappErr_t (*tsaCompletionFunc_t)(
   Int                    instId,
   UInt32                 flags,
   ptsaCompletionArgs_t   args
);
```

Fields

`instId`	Parent (creator) of the component layer calling the function.
`flags`	Use completion flags of type **tsaCompletionFlags_t** or cast to **UInt32**.
`args`	Pointer to completion function arguments of the type **tsaCompletionArgs_t**.

Description

The completion callback function is called by a component to report any specific completion to its parent. **flags** is the completion flag of type **tsaCompletionFlags_t** or cast to **UInt32**. The type is **UInt32** in the function definition to allow user-defined flags without casting.

args is a pointer to a **tsaCompletionArgs_t** structure, which contains the completion code specific to the component. Component-specific completion codes are described in the API documentation of the component.

tsaCompletionFlags_t

```
typedef enum {
   tsaCompletionFlagNone   = 0x00000000,
   tsaCompletionFlagStop   = 0x01000000,
} tsaCompletionFlags_t;
```

Description

This type is used by **tsaCompletionFunc_t** functions. **tsaCompletionFlagStop** is used by the default task when its stop sequence has been completed. See **tsaCompletionFunc_t** on page 173. **tsaCompletionFlagNone** is the default flag.

tsaCompletionArgs_t

```
typedef struct tsaCompletionArgs {
   Int       completionCode;
   Pointer   description;
} tsaCompletionArgs_t, *ptsaCompletionArgs_t;
```

Description

This type is used by **tsaCompletionFunc_t** functions. The **completionCode** identifies the completion reported by a component and is enumerated in the component's header file. The **description** field is specific to the individual component. See **tsaCompletionFunc_t** on page 173.

tsaDefaultDatainFunction

```
extern tmLibappErr_t tsaDefaultDatainFunction (
   Int                instId,
   UInt32             flags,
   ptsaDatainArgs_t   args
);
```

Parameters

`instId`	An opaque integer that is the parent ID of the component's AL layer. In fact, this opaque integer is a pointer to an instance variable structure whose first entry is a pointer to a default instance variable. See tsaDefaultInstVar_t on page 137.
`flags`	Use flags of type **tsaDatainFlags_t** or cast to **UInt32**.
`args`	Arguments of type **tsaDatainArgs_t**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instId** is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	Either **divp–>instSetup** or **args** is null.
`TMLIBAPP_ERR_NULL_DATAINFUNC`	Application installed a null datain callback function.
`TMLIBAPP_ERR_NULL_IODESC`	The specific **tsaInOutDescriptor_t** needed is null.
`TMLIBAPP_ERR_INVALID_CHANNEL_ID`	The channel ID specified in args is invalid for this component.
`TMLIBAPP_ERR_INVALID_COMMAND`	Command of packet received is invalid.
`TMLIBAPP_ERR_INVALID_FLAGS`	**flags** is invalid.
`TMLIBAPP_ERR_NOT_CACHE_ALIGNED`	Data buffers of packets created by the application or component is not cache aligned. Applies only to component that requests invalidation of packets. See tsaInOutDescSetupFlags_t on page 134.
`TMLIBAPP_ERR_IN_PLACE`	In-place component should not call this function with tsaDatainPutEmpty.
`TMLIBAPP_ERR_NULL_PACKET`	Packet to put on empty queue is Null.
`TMLIBAPP_NEW_FORMAT`	New format found in packet received.
`TMLIBAPP_PIN_STOPPED`	To notify component that this pin is stopped.
`TMLIBAPP_STOP_REQUESTED`	**divp–>taskstatus** is **TS_STOP_REQUESTED**.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system.

Description

Checks validity of **instId** (cast to **divp**) and **divp->setup** (cast to **disp**). Calls application-specific datain function if exists and returns. In the simple case, this function does the following. If **flags** contains **tsaDatainGetFull**, wait on full queue for a data packet, if **flags** contains **tsaDatainWait**. If **flags** contains **tsaDatainPutEmpty**, put data packet given in *args* on empty queue. However, in certain cases, other operations are performed. They are described in the following paragraph.

When **flags** contains **tsaDatainGetFull**, loops until a data packet is received (when flags contains **tsaDatainWait**) or returns during looping with a significant error code. If the task status is **STOP_REQUESTED**, it returns **TMLIBAPP_STOP_REQUESTED**, so as to break the component instance out of its running loop. If the component has a **waitSemaphore**, it waits on the semaphore instead of on the full queue. Checks the control queue if **flags** contains **tsaDatainCheckControl**. See **tsaDefaultControlioFunction** on page 187. If a data packet is received, returns the packet to the component instance, after invalidating if necessary and noting any format change. But, if the pin is stopped, puts the data packet on the emptyQueue, and returns **TMLIBAPP_PIN_STOPPED** if necessary. If an end of stream packet is received, calls the progress function to notify the application. Returns **TMLIBAPP_ERR_INVALID_COMMAND** if the packet type is not **tsaCmdDataPacket** or **tsaCmdWakeup**. Packet types are enumerated in **tsaDefaultControlMessage_t**. See **tsaDefaultControlMessage_t** on page 190.

When **flags** contains **tsaDatainPutEmpty**, **tsaDefaultDatainFunction** releases the format of the packet and puts the packet to the empty queue.

tsaDatainFunc_t

```
typedef tmLibappErr_t (*tsaDatainFunc_t)(
   Int                instId,
   UInt32             flags,
   ptsaDatainArgs_t   args
);
```

Fields

`instId`	Parent (creator) of the component layer calling the function.
`flags`	Use datain flags of type **tsaDatainFlags_t** or cast to **UInt32**.
`args`	Pointer to datain function arguments of the type **tsaDatainArgs_t**.

Description

The datain callback function is called by a component to access the full and empty queues of an input descriptor of the component. **flags** is the datain flag of type **tsaDatainFlags_t** or cast to **UInt32**. The type is **UInt32** in the function definition to allow user-defined flags without casting

args is a pointer to a **tsaDatainArgs_t** structure. The application should not install its own datain function, because **tsaDefaultDatainFunction** performs many operations depending on the state of the component instance and the input descriptor that are essential to the correct working of TSSA.

tsaDatainFlags_t

```
typedef enum {
   tsaDatainNone           = 0x00000000,
   tsaDatainGetFull        = 0x00000001,
   tsaDatainPutEmpty       = 0x00000002,
   tsaDatainWait           = 0x00000004,
   tsaDatainCheckControl   = 0x00000008,
} tsaDatainFlags_t;
```

Description

This type is used by **tsaDatainFunc_t** functions, particularly **tsaDefaultDatainFunction**. **tsaDatainGetFull** indicates the function to get a data packet from the full queue. **tsaDatainPutEmpty** indicates the function to put a data packet on the empty queue. **tsaDatainWait** can be combined with **tsaDatainGetFull** to wait until a data packet is available. **tsaDatainCheckControl** indicates the function to check the control queue after waiting on the full queue. See **tsaDefaultDatainFunction** on page 175.

tsaDatainGetFull and **tsaDatainPutEmpty** cannot be set together, however, one of them *must* be set.

tsaDatainArgs_t

```
typedef struct tsaDatainArgs {
   UInt32    inputId;
   Pointer   packet;
   UInt32    timeout;
} tsaDatainArgs_t, *ptsaDatainArgs_t;
```

Description

This type is used by **tsaDatainFunc_t** functions, particularly **tsaDefaultDatainFunction**. It contains the ID of the input channel to access, a pointer to the data packet to send or receive, and a timeout for queue access. See **tsaDatainFunc_t** on page 177.

tsaDefaultDataoutFunction

```
extern tmLibappErr_t tsaDefaultDataoutFunction (
   Int                  instId,
   UInt32               flags,
   ptsaDataoutArgs_t    args
);
```

Parameters

`instId`	Instance of a default instance variable structure. See **tsaDefaultInstVar_t** on page 137.
`flags`	Use flags of type **tsaDataoutFlags_t** or cast to **UInt32**.
`args`	Arguments of type **tsaDataoutArgs_t**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instId** is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	**divp->instSetup** or **args** is Null.
`TMLIBAPP_ERR_NULL_DATAOUTFUNC`	Application installed a Null dataout callback function.
`TMLIBAPP_ERR_NULL_IODESC`	The specific **tsaInOutDescriptor_t** needed is Null.
`TMLIBAPP_ERR_INVALID_CHANNEL_ID`	The channel ID specified in args is invalid for this component.
`TMLIBAPP_ERR_INVALID_COMMAND`	Command of packet received is invalid.
`TMLIBAPP_ERR_INVALID_FLAGS`	**flags** is invalid.
`TMLIBAPP_ERR_IN_PLACE`	In-place component should not call this function with **tsaDatainGetEmpty**.
`TMLIBAPP_ERR_NULL_PACKET`	Packet received from empty queue or packet to put on full queue is Null.
`TMLIBAPP_STOP_REQUESED`	**divp->taskstatus** is **TS_STOP_REQUESTED**.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system.

Description

Checks the validity of **instId** (cast to **divp**) and **divp->setup** (cast to **disp**). Calls application-specific dataout function if it exists in which case it returns. In the simple case, this function does the following. If **flags** contains **tsaDataoutGetEmpty**, this function waits on the empty queue for a data packet. If **flags** contains **tsaDataoutWait** it will be a blocking wait. If **flags** contains **tsaDatainPutFull**, this function puts the data packet given in

args to the full queue. However, in certain cases, other operations are performed, which is described below.

When **flags** contains **tsaDatainGetEmpty**, this function loops until a data packet is received (when **flags** contains **tsaDataoutWait**) or returns during looping with an error code. If the task status is **STOP_REQUESTED** and **flags** does not contain **tsaDatainIgnore-StopRequested**, this function returns **TMLIBAPP_STOP_REQUESTED**, in order to break the component instance out of its running loop. This function checks the control queue if flags contains **tsaDatainCheckControl**. See **tsaDefaultControlioFunction** on page 187. If a data packet is received, this function returns the packet to the component instance. It returns **TMLIBAPP_ERR_INVALID_COMMAND** if the packet type is not a **tsaCmdDataPacket** or **tsaCmdWakeup**. For Packet types, see **tsaDefaultControlMessage_t** on page 190.

When **flags** contains **tsaDatainPutFull**, this function claims the current format for the packet and puts it on the full queue. It increments **waitSemaphore** if it exists. If the pin is stopped and **flags** contains **tsaDataoutScheduleOnStop** it will call the OS sleep function to schedule out, and after the sleeping period of 1 OS tick, it puts the data packet on the emptyQueue, and returns.

tsaDataoutFunc_t

```
typedef tmLibappErr_t (*tsaDataoutFunc_t)(
   Int                 instId,
   UInt32              flags,
   ptsaDataoutArgs_t   args
);
```

Fields

`instId`	Parent (creator) of the component layer calling the function.
`flags`	Use dataout flags of type **tsaDataoutFlags_t** or cast to **UInt32**.
`args`	Pointer to dataout function arguments of the type **tsaDataoutArgs_t**.

Description

The dataout callback function is called by a component to access the full and empty queues of an output descriptor of the component. **flags** is the dataout flag of type **tsaDataoutFlags_t** or cast to **UInt32**. The type is **UInt32** in the function definition to allow user-defined flags without casting.

args is a pointer to a **tsaDataoutArgs_t** structure. The application should not install its own dataout function, because **tsaDefaultDataoutFunction** performs many operations depending on the state of the component instance and the output descriptor that are essential to the correct working of TSSA.

tsaDataoutFlags_t

```
typedef enum {
   tsaDataoutNone                = 0x00000000,
   tsaDataoutGetEmpty            = 0x00000001,
   tsaDataoutPutFull             = 0x00000002,
   tsaDataoutWait                = 0x00000004,
   tsaDataoutCheckControl        = 0x00000008,
   tsaDataoutScheduleOnStop      = 0x00000010,
   tsaDataoutIgnoreStopRequested = 0x00000020,
   tsaDataoutIgnorePutEmpty      = 0x00000040,
} tsaDataoutFlags_t;
```

Description

This type is used by **tsaDataoutFunc_t** functions, particularly **tsaDefaultDataoutFunction**. **tsaDataoutGetEmpty** indicates the function to get a data packet from the empty queue. **tsaDataoutPutFull** indicates the function to put a data packet on the full queue. tsaDatainWait can be combined with **tsaDataoutGetEmpty** to wait until a data packet is available. **tsaDatainCheckControl** indicates the function to check the control queue after waiting on the empty queue. **tsaDataoutScheduleOnStop** indicates the function to sleep if the pin is stopped to prevent the component task from always running. **tsaDataoutIgnoreStopRequested** indicates the function to ignore the task status stop request and continue without returning **TMLIBAPP_STOP_REQUESTED**. See **tsaDataoutFunc_t** on page 181.

tsaDataOutPutEmpty iindicates that a packet must be put on the empty queue. This allows components to get rid of packets during stop while there is no format installed on the queue or when there is no data in the packet.

tsaDataoutGetEmpty and **tsaDataoutPutFull** cannot be set together, however, one of them *must* be set.

tsaDataoutArgs_t

```
typedef struct tsaDataoutArgs {
   UInt32    outputId;
   Pointer   packet;
   UInt32    timeout;
} tsaDataoutArgs_t, *ptsaDataoutArgs_t;
```

Description

This type is used by **tsaDataoutFunc_t** functions, particularly **tsaDefaultDataoutFunction**. It contains the ID of the output channel to access, a pointer to the data packet to send or receive, and a timeout for queue access. See **tsaDataoutFunc_t** on page 181.

tsaDefaultMemallocFunction

```
extern tmLibappErr_t tsaDefaultMemallocFunction (
   Int                  instId,
   UInt32               flags,
   ptsaMemallocArgs_t   args
);
```

Parameters

`instId`	Instance of a default instance variable structure. See **tsaDefaultInstVar_t** on page 137.
`flags`	Flags to be passed to application-specific memalloc function, if exists.
`args`	Arguments to passed to application-specific memalloc function, if exists.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instId** is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	**divp->instSetup** is null.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Not enough memory.

Description

This function is installed as the default memory allocation function used by TSSA components.
Checks validity of **instId** (cast to **divp**) and **divp->setup**. If the user has specified an application-specific memalloc function, it is dispatched from here. Otherwise, calls **_cache_malloc** to allocate the memory requested to assure 64-byte alignment and initializes memory to 0 with **memset**. Then, returns memory allocated in **args–>memHandle**.

The alignment field of the args structure is passed to **_cache_malloc** as the second parameter, specifying cache set. Set the alignment field to -1 if you do not want to specify the cache set.

tsaMemallocFunc_t

```
typedef tmLibappErr_t (*tsaMemallocFunc_t)(
   Int                  instId,
   UInt32               flags,
   ptsaMemallocArgs_t   args
);
```

Fields

`instId`	Parent (creator) of the component layer calling the function.
`flags`	Flags for memalloc function.
`args`	Pointer to memalloc function arguments of the type **tsaMemallocArgs_t**.

Description

The memalloc callback function is called by a component to allocate a certain size of memory. The type for **flags** is **UInt32** in the function definition to allow user-defined flags without casting.

tsaMemallocArgs_t

```
typedef struct tsaMemallocArgs {
   Int       sizeInBytes;
   Int       alignment;
   Pointer   memHandle;
} tsaMemallocArgs_t, *ptsaMemallocArgs_t;
```

Description

This type is used by **tsaMemallocFunc_t** functions. **sizeInBytes** indicates the size of the memory to be allocated. alignment indicates the alignment required for memory allocation. **memHandle** is the pointer to the allocated memory to be returned. See **tsaMemallocFunc_t** on page 184.

tsaDefaultMemfreeFunction

```
extern tmLibappErr_t tsaDefaultMemfreeFunction (
   Int                  instId,
   UInt32               flags,
   ptsaMemfreeArgs_t    args
);
```

Parameters

`instId`	Instance of a default instance variable structure. See **tsaDefaultInstVar_t** on page 137.
`flags`	Flags to be passed to application-specific memfree function, if it exists.
`args`	Arguments to passed to application-specific memfree function, if it exists.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instId** is not a valid default instance variable structure. Cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	**divp->instSetup** is null.

Description

Checks validity of **instId** (cast to **divp**) and **divp->setup**. Calls application-specific memfree function if exists and returns. Otherwise, calls **_cache_free** to free the memory in **args–>memHandle** if exists.

tsaMemfreeFunc_t

```
typedef tmLibappErr_t (*tsaMemfreeFunc_t)(
   Int                 instId,
   UInt32              flags,
   ptsaMemfreeArgs_t   args
);
```

Fields

`instId`	Parent (creator) of the component layer calling the function.
`flags`	Flags for memfree function.
`args`	Pointer to memfree function arguments of the type **tsaMemfreeArgs_t**.

Description

The memfree callback function is called by a component to free memory. The type for **flags** is **UInt32** in the function definition to allow user-defined flags without casting.

tsaMemfreeArgs_t

```
typedef struct tsaMemfreeArgs {
   Pointer   memHandle;
} tsaMemfreeArgs_t, *ptsaMemfreeArgs_t;
```

Description

This type is used by **tsaMemfreeFunc_t** functions. **memHandle** is the pointer to the memory to be freed. See **tsaMemfreeFunc_t** on page 186.

tsaDefaultControlioFunction

```
extern tmLibappErr_t tsaDefaultControlioFunction (
   Int                  instId,
   tsaControlFlags_t    flags,
   ptsaControlArgs_t    args
);
```

Parameters

`instId`	Instance of a default instance variable structure. See **tsaDefaultInstVar_t** on page 137.
`flags`	Flags to be passed to application-specific controlio function, if it exists.
`args`	Arguments to be passed to an application-specific controlio function, if it exists.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	**instId** is not a valid default instance variable structure. cast to **divp**.
`TMLIBAPP_ERR_INVALID_ARGS`	**args** is Null.
`TMLIBAPP_ERR_INVALID_FLAGS`	**flags** is invalid.
`TMLIBAPP_ERR_OS_ERR`	Indicates an error from the operating system.

Description

This function is used internally by the TSA defaults to implement the queue based instance config mechanism. The function checks the validity of **instId** (cast to **divp**) and **divp->setup**. If **flags** contains **tsaControlAppToComponent**, it attempts to get a packet from the command queue. If **flags** contains **tsaControlWait, this is a blocking queue_receive**. If **flags** contains **tsaControlComponentToApp**, it sends a packet containing the return value from the command operation on the response queue. Packet types are enumerated in **tsaDefaultControlMessage_t**. See page 190.

The controlio function is called by **tsaDefaultDatainFunction** and **tsaDefaultDataoutFunction** when it is requested to check the control queue. The sequence for checking and executing the command is as follows. If no control descriptor was installed, the datain or dataout function returns **TMLIBAPP_OK**. The validity of **instId** (cast to **divp**) and **divp->setup** (cast to **disp**) are checked. Then **disp->controlFunc**, which is **tsaDefaultControlioFunction** by default, is called with **flags** containing **tsaControlAppToComponent**. If the queue is empty or if a timeout occurred, it returns **TMLIBAPP_OK**. If a packet is received, it checks the associated command in **cargs.command**. If the command is **tsaCmdStatus**, it returns an acknowledgement by calling **disp->controlFunc** with **flags** containing **tsa-**

ControlComponentToApp. For all other commands, it calls the AL instance config function and the sends an acknowledgement with its return value. If the AL instance config function is null, it returns **TMLIBAPP_ERR_NULL_CONTROLFUNC**.

tsaControlFunc_t

```
typedef tmLibappErr_t (*tsaControlFunc_t)(
   Int                 instId,
   tsaControlFlags_t   flags,
   ptsaControlArgs_t   args
);
```

Fields

`instId`	Parent (creator) of the component layer calling the function.
`flags`	Controlio flags of type **tsaControlFlags_t**.
`args`	Pointer to controlio function arguments of the type **tsaControlArgs_t**.

Description

The controlio callback function is called by a component from **tsaDefaultDatainFunction** or **tsaDefaultDataoutFunction** to check the command queue or reply in the response queue.

tsaControlFlags_t

```
typedef enum {
   tsaControlNone            = 0x00000000,
   tsaControlAppToComponent  = 0x00000001,
   tsaControlComponentToApp  = 0x00000002,
   tsaControlWait            = 0x00000004,
} tsaControlFlags_t;
```

Description

This type is used by **tsaControlFunc_t** functions. **tsaControlAppToComponent** indicates the function to get a command packet from the command queue. **tsaControlComponentToApp** indicates the function to send a response packet to the response queue. **tsaControlWait** can be combined with **tsaControlAppToComponent** to wait until a command packet is available. See **tsaControlFunc_t** on page 188.

tsaControlAppToComponent and **tsaControlComponentToApp** cannot be set together, however, one of them *must* be set.

tsaControlArgs_t

```
typedef struct tsaControlArgs_t {
   UInt32          command;
   Pointer         pamameter;
   tmLibappErr_t   retval;
   UInt32          timeout;
} tsaControlArgs_t, *ptsaControlArgs_t;
```

Description

This type is used by **tsaControlFunc_t** functions. It contains the command to the component instance, optional parameters, the return value from the command operation, and a timeout for queue access. See **tsaControlFunc_t** on page 188.

tsaDefaultControlMessage_t

```
typedef enum {
   tsaCmdDataPacket    = 0x00,
   tsaCmdWakeup        = 0x10,
   tsaCmdEndOfStream   = 0x11,
   tsaCmdStatus        = 0x21,
   tsaCmdAcknowledge   = 0x22,
   tsaCmdUserBase      = 0x50,
} tsaDefaultControlMessage_t;
```

Description

This type is used by **tsaDatainFunc_t**, **tsaDataoutFunc_t**, and **tsaControlFunc_t** functions. It indicates the packet type of a packet received from a queue. **tsaCmdDataPacket** indicates a data packet sent and received by **tsaDatainFunc_t** and **tsaDataoutFunc_t**. **tsaCmdWakeup is used by the tsaDefaults to cause a component blocked on its data queues to wake up and check for activity. tsaCmdEndOfStream can be used to initiate a system stop when the data stream has ended**. See **tsaDefaultStop** on page 160. **tsaCmdStatus**, and **tsaCmdAcknowledge** are used by **tsaControlFunc_t** to specify the command and acknowledgement of the associated packet. **tsaCmdUserBase** can be used as a base code by the components or application to create customized packet types.

