# *Book 3—Software Architecture*

## Part A:

# Foundation

TriMedia

Version 2.1

# Table of Contents

Chapter 1

# TriMedia Software Architecture Overview

## Introduction

The goal of TriMedia Software Architecture (TSA) is to promote interoperability and reusability of components, thereby enabling seamless collaboration between application programmers and component developers.

There are many aspects to the architecture. Some portions that are of interest to one developer may not be of interest to another. The overriding concern has been to keep the code base flexible and lightweight so that all needs are met. We hope that the TSA guidelines that have evolved from our experience will save you considerable time.

### What is TSA?

The TriMedia C compiler and related tools, including the TriMedia debugger, form the backbone of the software architecture. Other parts of TSA include the basic C library, host interface libraries, device libraries, pSOS (the real-time operating system), and TSSA (TriMedia Streaming Software Architecture). All libraries are provided as binaries (both big- and little-endian), with header files, documentation, and sample programs.

Key components of the TSA are covered in the various chapters of this reference.

## The Development Environment

At its core, the TriMedia development environment is modeled on a traditional command-based C development environment. Books 4 through 7 provide more information about the TriMedia software development environment (SDE).

The **make** utility is a very useful part of the command-line development environment. You can customize the provided makefile examples to suit your particular needs. Furthermore, you can choose any particular editor to use with the TriMedia SDE. Because TriMedia does not provide an integrated development environment (IDE), users in the past have created their own development environments in UNIX or in Windows using an editor such as Codewright™.

For those users who prefer the Metrowerks CodeWarrior IDE, this TriMedia release includes TriMedia tools as plugins for CodeWarrior. The plugins are compatible with Win95, Win98, WinNT and MacOS CodeWarrior Professional Release 4 or later.

## Universal Concepts

A few basic ideas, presented here, serve as a foundation for TSA.

### Layers

The TriMedia software architecture can be divided into a number of *layers*. A typical, and somewhat elementary, use of layers, illustrated in Figure 1, is described in more detail later in this book. A knowledge of these layers is essential for a complete understanding of TSA.

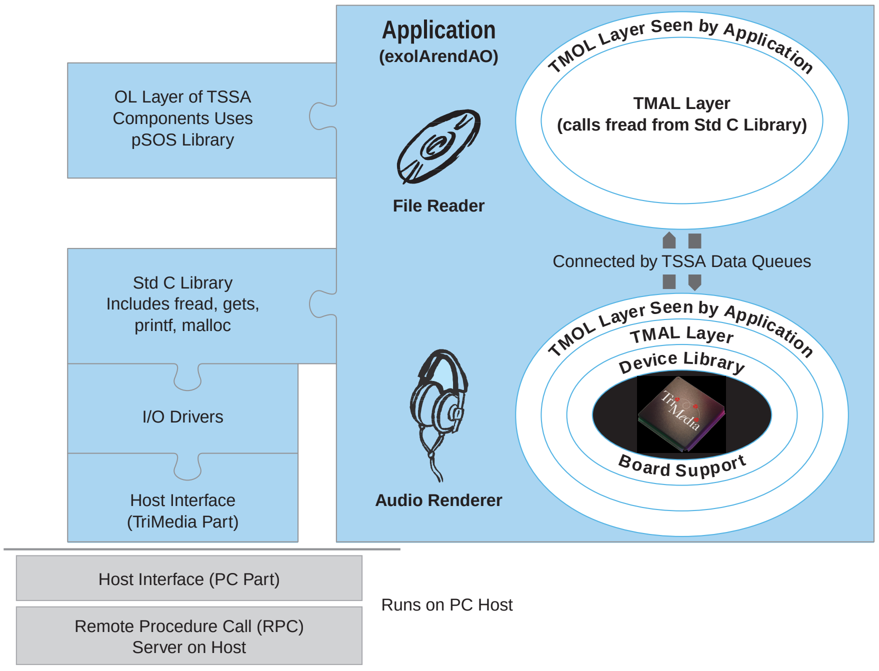
Application
(exolArendAO)
TMOL Layer Seen by Application
TMAL Layer
(calls fread from Std C Library)
OL Layer of TSSA Components Uses pSOS Library
File Reader
Connected by TSSA Data Queues
Std C Library Includes fread, gets, printf, malloc
I/O Drivers
Host Interface (TriMedia Part)
Audio Renderer
TMOL Layer Seen by Application
TMAL Layer
Device Library
Board Support
Host Interface (PC Part)
Remote Procedure Call (RPC) Server on Host
Runs on PC Host

**Figure 1** How Layers Work With a Sample Application

**Table 1** TriMedia Application Hierarchy

| Layer | Notes |
|---|---|
| Application | Controls the system.<br>Has a user interface.<br>Uses the functionality of libraries. |
| pSOS | The real-time operating system, as a library.<br>Complete RTOS developed by Integrated Systems.<br>Provides task control, scheduling, and messaging. |
| TSSA<br>OL Layer | Operating-system-dependent streaming software layer.<br>Adheres to standard interface. |
| TSSA<br>AL Layer | Operating-system-independent streaming software layer.<br>Adheres to standard interface.<br>Defines protocols for the exchange of multimedia data. |
| Device Library | Lowest level software interface to the hardware.<br>The board support package defines a way to control off chip peripherals like A/D/A converters used by the audio and video systems. |
| Host Interface | The various layers of the host interface, including the remote procedure call server used to implement fread and printf are illustrated. |

Refer to Chapter 3 for an introduction to host interfaces for TriMedia.

Refer to Chapter 5 for an introduction to device libraries on TriMedia.

Refer to Chapter 6 for an introduction to using pSOS on TriMedia.

Refer to Chapter 7 (Part B) for an introduction to the streaming software architecture.

## Naming Conventions

The root path of the TriMedia application library package is $(TAS).

The file $(TCS)/include/tmlib/tmtypes.h defines a number of fundamental types that are used throughout TriMedia code. This file defines types such as **UInt32** and **UInt16**. This file serves as a cornerstone of future variants of the architecture.

A few naming conventions are used throughout the system:

- The suffix **_t** denotes a type definition.
- The prefix **p** often denotes a pointer.
- Variable and function names in TriMedia libraries do not include underscores. Instead, words in names are concatenated. The first letter is always lower case; every subsequent word starts with a capital letter. Two examples are **tmalAdecAc3Stop** or **tsaClockOpen**. The word **tmVersion_t** is a data type; a variable of type **ptmVersion_t** is a pointer.

- Prefixes such as **dw** that are commonly used in Windows software are not used in TriMedia software.
- Function names are prefixed with **tm**, or **tsa** to show their link to TriMedia.
- The prefix **tmol** applies to function libraries (at the OL layer) that depend on a real-time multitasking operating system such as pSOS. Chapter 7, *TSSA Essentials,* explains in detail the meaning of "OL."
- The prefix **tmal** applies to function libraries (at the AL layer) that do not depend on a real-time multitasking operating system. Chapter 7, *TSSA Essentials,* explains in detail the meaning of "AL."
- We have guarded against name space pollution. Library functions are explicitly declared static or external. External functions are prototyped in the header file that represents the module. Other functions that are not static are prefixed with an underscore.
- The prefix **dev** is a generic device name, used in these manuals, for which a specific device name can be substituted. Device libraries provide a software interface for the on-chip TriMedia peripherals. Specific device names include **tmAO**, **tmAI**, **tmVO**, **tmVI**, **tmSSI**, **tmINT**, **tmTIM**, and others.
- The term **COM** is a generic component name, used in these manuals, for which specific component names can be substituted. Examples of specific component names in functions are **tmolFread** and **tmolArendAO**. This convention may also be written **Com** or **com**, as appropriate.

## Memory Management

The TCS standard C library includes an implementation of **malloc** and **free**. Blocks are allocated in multiples of 4 bytes.

- The pSOS memory manager allows the creation of "regions." Regions can separate the memory usage of two components. For example, the use of two regions would ensure that a component requiring dynamic reallocation of small blocks of memory would not fragment the larger blocks required by another program.

  TSSA components address this issue by allowing you to install custom memory managers on a per-module basis.
- When pSOS is installed, **malloc** and **free** can be mapped to the pSOS functions that allocate from region zero. Or, at your discretion, they can be mapped to the TCS **malloc** and **free**. (See the discussion of **TCS_MALLOC_USE** on page 78 of Chapter 6, *pSOS+™ Real-Time Operating System*.)
- The functions **cache_malloc** and **cache_free** are built on top of **malloc** and **free**, respectively. They will be affected by redirection by the user.

## Error Reporting

TriMedia libraries report unique 32-bit error codes. The error values returned by TriMedia libraries can be decoded according to the rules that follow.

Device library functions return errors typed as **tmLibdevErr_t**.

Application libraries return the type **tmLibappErr_t**.

Practically, these codes are unsigned 32-bit integers. The fundamental definitions are contained in $(TCS)/tm1/tmLibdevErr.h for device libraries and $(TAS)/include/tmLibappErr.h for application libraries.

In addition to the return of error codes, Philips uses an assertion mechanism liberally to debug TriMedia libraries. Unless the library in question has been compiled with the flag **-DNO_DEBUG**, assertions are used to check many values. When an assertion is triggered, the file name, the line number, and the error code are reported both to **STDERR**, and to the **DP** buffer. The assertion calls **Exit**, because triggered assertions are fatal errors. Philips *highly recommends* that you build your programs with assertions enabled. Use the "**_g**" version of the libraries.

## Error Decoding

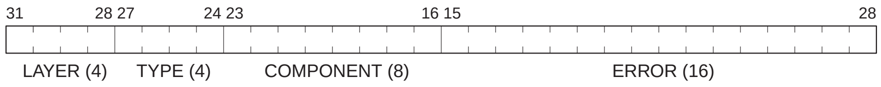

**Figure 2** Error Register

- Bits 28–31 identify the layer where the error originated, as defined in $(TCS)/tm1/mLibdevErr.h:

```
0xxxxxxx:  Device Library
1xxxxxxx:  AL Layer
2xxxxxxx:  OL Layer
3xxxxxxx:  Application Layer
```

- Bits 24–27 describe the type of component that reported the error. Although it is possible to add other error types, it should not be necessary for most applications, as these error types were designed to be generic. However, if absolutely necessary, the procedure for adding other error types is described in tmLibdevErr.h. Examples of existing error types include:

```
x0xxxxxx:  Generic (unspecified)
x1xxxxxx:  System
x2xxxxxx:  Graphics
x3xxxxxx:  Video
x4xxxxxx:  Audio
x5xxxxxx:  Communications
xFxxxxxx:  Other (used by the Board Support Package)
```

- Bits 16–23 identify the component from which the error was returned.

- Bits 0–15 identify the specific error code.

Error codes for the device libraries are defined in tmLibdevErr.h. Application library components define specific error codes in their respective header files. However, a significant number of commonly used error codes are defined in tmLibappErr.h.

## Examples of Errors

- **0x11000002** is a system error reported by the AL layer. The **2** is defined **TMLIBAPP_ERR_INVALID_SETUP** in tmLibappErr.h. This might mean, for example, that a setup structure passed to an instance setup function was **NULL** or contained invalid values.
- **0x21000076** is a system error returned from the OL layer. The **76** is defined **TMLIBAPP_ERR_FORMAT_NEGOTIATE_DATASUBTYPE**. This error is returned by the format manager to indicate that two components do not share a common data subtype.
- **0x11010002** is an AL layer system error reported by component 1. TmLibappErr.h tells us that component 1 is the file reader. TmalFread.h informs us that error 2 is **FR_ERR_CLOSE_FAILED**. This error code is returned when an attempt to close a file fails.

## Device Drivers

Device drivers are an important part of TSA. Although TriMedia does not explicitly use a traditional device-driver architecture, each device driver in TriMedia does include elements of a traditional device driver. Note that pSOS device drivers are neither provided nor used by TriMedia.

Even though a TriMedia device driver provides the same services as a traditional device driver, it does so through a different interface. The difference is explained below.

A traditional device driver interface includes the following functions:

**open close read write setup**

The problem with this interface is that the **read** and **write** functions imply layers of buffering that can be problematic in a data-intensive multimedia system. Instead, TriMedia splits the device driver into a number of levels. The lowest level (known as the device library layer) provides **open**, **close** and **setup** functions, but it does not specify **read** and **write** functions. The data transfer mechanism is left up to a higher layer of the software.

The on-chip TriMedia peripherals provide interrupt vectors. Interrupts are the most efficient way to use the peripherals. The layer above the device driver is responsible for installing interrupt service routines (ISRs). This layer is the “AL,” which will be explained in Chapter 7, *TSSA Essentials*. The ISRs can be written in C, using the **TCS_handler** pragma. Because the AL layer does not depend on an operating system, it can be used with or without an OS. When it is used with an OS, a system of callback functions allows operating system calls to be made at the appropriate times. In this way, an application that uses the pSOS operating system will use an OL layer library as a device driver. The standard OL layer interface includes equivalents of the traditional **open**, **close**, and **setup** routines. The read and write functionality is implemented using the datain and dataout callback functions that are specified by TSSA.

## The Libraries

The remainder of this book explains each TriMedia library in detail. The TriMedia libraries are presented in the following order.

### Standard C Library

The TriMedia Software Architecture provides a version of the standard C library, complete with read and write functions such as **fread** and **gets**. The I/O functions have been implemented on top of an easy-to-replace, portable layer. Chapter 2 describes the standard C library.

### Host Interface Libraries

Because TriMedia is often used in conjunction with Windows and Macintosh personal computers, Philips has developed host interface libraries to support Windows 95, Windows NT, and MacOS. As the programs and the interfaces have been ported and developed, a consistent and portable host architecture has evolved. The Windows NT version of the host interface best illustrates this. Each of the host interface libraries implements downloading, program startup, and runtime communication services. The host libraries are described in Chapter 3, *Host Windows Interfaces*. Book 5, *System Utilities*, presents detailed API references for the host interfaces.

Downloader and dynamic loader libraries are also available. These are described in Chapter 11, *Linking TriMedia Object Modules* of Book 4, *Software Tools*, Part B.

### Device Libraries

Because the TriMedia family of processors includes numerous on-chip peripherals, a set of libraries (known collectively as libdev) is provided to facilitate the use of these peripherals. The device libraries are described in Chapter 5, *Device Libraries*. Detailed API references of device libraries are located in Books 5–9. Note that the device libraries do not dictate any data transfer mechanism, and do not depend on any operating system services.

## The pSOS Operating System

TSA draws a very clear line between components that depend or do not depend on an operating system. Although pSOS is the operating system supported and used by TSA, it is used through an operating system abstraction layer that allows porting to another real-time operating system. Chapter 6, *pSOS+™ Real-Time Operating System* covers many details of the TriMedia version of pSOS. For more general information, refer to the *pSOS Programmer's Reference* and *pSOS System Concepts* manuals, which are shipped with the TriMedia SDE.

Services provided by pSOS include tasks, queues, and semaphore management. These services form the basis of the TriMedia Software Streaming Architecture (TSSA). The pSOS system also provides a number of other supporting libraries. At least one of these, the pNA networking stack, is available on TriMedia.

## TriMedia Streaming Software Architecture (TSSA)

TSSA is the highest level of the TriMedia Software Architecture. It provides a method of constructing and connecting autonomous components that "stream" data between them. The basic developer's kit includes a core of TSSA components in the form of libraries, some with accompanying source code, and with a number of examples of their use. The TSSA framework was used to construct the HDTV decoder, which (along with several other TSSA-compliant libraries) are available as the TriMedia DTV software developer's kit. Books 5-9 gives detailed API references.

An introduction to TSSA can be found in Chapter 7, *TSSA Essentials*.

Chapter 2

# Standard C Library

## Standard C Library

A fundamental component of the TriMedia software architecture is the standard C library. The standard C library, as described in stdlib.h, provides a number of ANSI standard services. The standard C library, libstd.a, is linked to hosted applications by default. (Of course, it is not linked if it is not used.)

The standard C library includes the **fread**, **fwrite**, **printf**, and **scanf** functions. The availability of these functions can be a great aid in the development and testing of multimedia applications. These functions are implemented on top of a driver layer. This chapter gives an overview of that driver layer. The driver layer can interface to a host processor or it can be used in a stand-alone system that has access to I/O facilities.

This chapter is of particular importance to programmers who want to port the driver interface to another host.

## Standard C Host Interface

The Standard C library, the Remote Procedure Call (RPC) Client, and the RPC Server form the TriMedia Standard C Host Interface. See Figure 3 below.

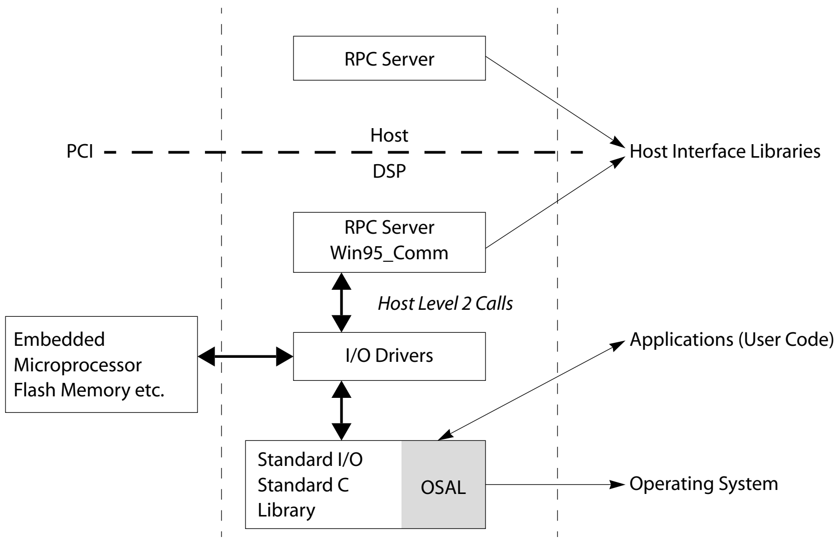

**Figure 3** Standard C Host Interface

The Standard C Host Interface component of the TriMedia Software Architecture provides basic I/O capabilities to TriMedia applications. It isolates the host-specific I/O functions that are necessary for the system library to work in a host-independent way. The Standard C Host Interface is independent of your choice of host and operating system, making it possible to write basic tests and demonstration programs in a portable fashion. In each case, the dependency is removed through an “abstraction layer” which is really just a clearly declared set of functions that must be available to the Standard C library. This approach to abstraction is easily extended to applications.

The Standard C Host Interface is modeled as an RPC interface that sends service requests in a way that is synchronous with the task or application that uses it. From a task’s point of view, the request has been serviced as soon as the corresponding function call terminates. However, the Standard C Host Interface can silently yield the processor to other tasks if the request requires waiting. The Standard C Host Interface achieves this by obtaining the current scheduler functions through the Application Model interface.

For example, a task can read a large buffer from a file by issuing the following call:

```
read( datafile, buffer, 1000000 );
```

Not only does the resulting **read** request take some time to reach the host, but letting this host read a megabyte of data into the user-provided buffer causes additional delay. Because TriMedia would otherwise spend idle cycles waiting for completion, the current task is descheduled, causing other tasks (when available) to continue while the reading task waits.

The following HostCall service requests are supported by the Standard C Host Interface:

```
HostCall_OPEN
HostCall_FSTAT
HostCall_ISATTY
HostCall_READ
HostCall_LSEEK
HostCall_WRITE
HostCall_CLOSE
HostCall_UNLINK
HostCall_MKTEMP
HostCall_GETENV
HostCall_LINK
HostCall_TIME
HostCall_SOCK_SEND
HostCall_SOCK_RECV
HostCall_SOCK_STATUS
HostCall_SOCK_DATA
HostCall_TMPNAM
HostCall_FCNTL

HostCall_SYSTEM
HostCall_OPENDLL
HostCall_ARGV_ARGC_INFO
HostCall_GET_ARGUMENT_STRING
HostCall_EXIT
```

All but the last five service requests correspond to the POSIX.1 functions that require host communication.

The I/O drivers component of the Standard C Host Interface maintains a set of I/O drivers. An I/O driver is a collection of file I/O functions including a filename recognition function. The following is an example:

```
struct UID_Driver {
   IOD_RecogFunc     recog;
   IOD_InitFunc      init;
   IOD_TermFunc      term;
   IOD_OpenFunc      open;
   IOD_CloseFunc     close;
   IOD_ReadFunc      read;
   IOD_WriteFunc     write;
   IOD_SeekFunc      seek;
   IOD_IsattyFunc    isatty;
   IOD_FstatFunc     fstat;
   IOD_FcntlFunc     fcntl;
   IOD_OpenDllFunc   open_dll;
   UID_Driver        next;
};
typedef struct UID_Driver *UID_Driver;
```

Such a driver can be installed and uninstalled at any time. Upon opening a file, the collection of currently installed drivers is traversed in reverse order of installation, until the corresponding filename recognition function returns **TRUE**. This driver is selected for the file to be opened, and the corresponding functions (starting with the **open** function) are the ones used for this file.

The RPC Client/Server component of the Standard C Host Interface can be as simple or as complex as you want. In the case of a Windows 95 or Macintosh Operating System (MacOS) host, the RPC module consists of a client running on TriMedia, as well as a server running under the Windows 95 or Mac operating system. The two communicate through the TriMedia manager's host API. The interface uses the following four functions:

```
HostCall_send
HostCall_notify
HostCall_init
HostCall_term
```

The protocol by which a HostCall service should be requested from the host is as follows:

1. The application prepares a **HostCall_command** buffer, and fills it with the proper function code (see Figure 4) and corresponding parameters. The **HostCall_command** buffer is also used for returning the service results, so it should not be modified or deallocated before the request is serviced.
2. The application then issues a call to the **HostCall_send** function, with a pointer to the buffer as an argument.
3. After completion of this call, the service has either completed successfully or failed. You can find information on this in the status field, which is set to either **HostCall_DONE** or **HostCall_ERROR**. An error indicates a failure in the host communication, not an error in the execution of the requested service itself. For example, the reason for a failure to open a file is stored in **errno**, as is conventional.

**Note**
This sequence is implemented in the appropriate library calls and, therefore, should not be user-visible. Describing it provides the necessary clarification for porting the HostCall component to another host.

The **_HostCall_send** function, which is responsible for achieving the required synchronization using the AppModel interface, is host-independent and not subject to porting. Its implementation, shown in Figure 4, is part of the TriMedia runtime system.

**_HostCall_send** makes use of the host-specific **_HostCall_host_send** function, which is the function subject to porting. Note that **_HostCall_send** allows instantaneous serving, in which case no need for waiting exists.

You can use this option when no real host interaction is needed for servicing the request. A simple example is the shortcut of a file write of zero bytes. If waiting is required, **_HostCall_send** uses the Application Model interface.

The **_HostCall_host_send** function eventually calls the **_HostCall_notify** function to report termination of the service request. Results of the request are returned in the command buffer, and the notification function resumes the waiting requester.

```
void _HostCall_send( HostCall_command  *command ){
   command->requester              = AppModel_current_thread;
   command->status                 = HostCall_BUSY;
   command->notification_status    = HostCall_BUSY;
   command->termination_handler    = NULL;
   command->returned_errno         = 0;

   _HostCall_host_send( command );

   if( command->status == HostCall_BUSY ){
      AppModel_suspend_self();
      if( command->termination_handler ){
         command->termination_handler(command);
      }
      command->status = command->notification_status;
   }
   if( command->returned_errno ){
      errno = command->returned_errno;
   }
}
void _HostCall_notify( HostCall_command  *command ){
   AppModel_resume( command->requester );
}
```

**Figure 4** Implementation of _HostCall_send and _HostCall_notify

Porting to a new host involves the following steps:

1. A function with the following prototype must be defined:

```
void _HostCall_host_send ( HostCall_command  *command );
```

2. A call to this function must complete in a “very short” amount of time, having assigned one of three **HostCall_status** values in the **status** field of the command buffer:

— **HostCall_ERROR**: The service request has failed. Note that this only applies to the host communication. The service itself probably was not tried because of (for example) capacity problems of the host communication channel.

— **HostCall_DONE**: The service has already been provided. No waiting is needed. The results of the request are returned in command buffer.

— **HostCall_BUSY**: The host has accepted the request and "promised" to eventually try to service it, after which it will call the **_HostCall_notify** notification function, with the command buffer as argument. In this case, depending on success or failure, the **notification_status** field is set to either **HostCall_ERROR** or **HostCall_DONE**. Again, this status only provides information on success of host communication. Errors in the requested services themselves are reported in the **returned_errno** field.

3. **_HostCall_host_send** is responsible for endianness conversion, the mapping of non-standard **errno** values, and cache coherency.

## File I/O Drivers

Header file tmlib/IODrivers.h defines routines that allow a program to install file i/o drivers. A *file I/O driver* provides access to file manipulation functions, either through the usual system call functions (**open**/**read**/**write**/**close** and others) or through their standard C library counterparts (**fopen**/**fread**/**fwrite**/**fclose** and others), which the standard library implements using the underlying system calls.

For a program running in a hosted environment, either under the simulator **tmsim** or on hosted TM hardware, program startup installs a file i/o driver that performs file i/o using host routines. Thus, opening a file in a TriMedia program in a hosted environment will call the **open** routine of the host. For a program running in a nohost environment, no file i/o drivers are installed by default.

A program can install additional i/o drivers using **IOD_install_driver**, as defined in tmlib/ IODrivers.h:

```
UID_Driver IOD_install_driver(
   IOD_RecogFunc     recog,
   IOD_InitFunc      init,
   IOD_TermFunc      term,
   IOD_OpenFunc      open,
   IOD_OpenDllFunc   open_dll,
   IOD_CloseFunc     close,
   IOD_ReadFunc      read,
   IOD_WriteFunc     write,
   IOD_SeekFunc      seek,
   IOD_IsattyFunc    isatty,
   IOD_FstatFunc     fstat,
   IOD_FcntlFunc     fcntl,
   IOD_StatFunc      stat
);
```

The TriMedia Compilation System makes no assumptions about the format of the supported file system.  In particular. it has no knowledge of the format of file names.

Instead, it uses the *recognition functions* **recog** of the currently installed I/O drivers to determine whether a file name is recognized by a i/o driver. When a program performs an **open** call, the TCS run-time support calls the recognition functions of each installed I/O driver, latest-installed first, until it finds a recognition function that recognizes the file name (i.e. a **recog** function which returns TRUE). Then the run-time support uses the corresponding driver functions to perform all subsequent operations on the file. The recognition function for the default host system file I/O driver always returns TRUE, so it recognizes all host system file names. Of course, a subsequent **open** call will fail if the host system **open** fails, for example because the file does not exist.

The **init** member defines an initialization function that is called when the driver is installed, and the **term** member defines a function that is called if the driver is uninstalled using **IOD_uninstall_driver**. The **open**, **close**, **read**, **write**, **seek**, **isatty**, **fstat**, **fcntl**, and **stat** functions provide the standard POSIX.1 system call functionality for the driver. The C library documentation gives a brief synopsis of the usage of each of these calls, and POSIX.1 describes them in detail. Finally, the **open_dll** function opens a dynamic linked library (.dll).

The program can also use the slightly more general **IOD_install_fsdriver()** call:

```
UID_Driver IOD_install_fsdriver
    (
        IOD_RecogFunc     recog,
        IOD_InitFunc      init,
        IOD_TermFunc      term,
        IOD_OpenFunc      open,
        IOD_OpenDllFunc   open_dll,
        IOD_CloseFunc     close,
        IOD_ReadFunc      read,
        IOD_WriteFunc     write,
        IOD_SeekFunc      seek,
        IOD_IsattyFunc    isatty,
        IOD_FstatFunc     fstat,
        IOD_FcntlFunc     fcntl,
        IOD_StatFunc      stat,
        IOD_SyncFunc      sync,
        IOD_FSyncFunc     fsync,
        IOD_UnlinkFunc    unlink,
        IOD_LinkFunc      link,
        IOD_MkdirFunc     mkdir,
        IOD_RmdirFunc     rmdir,
        IOD_AccessFunc    access,
        IOD_OpendirFunc   opendir,
        IOD_ClosedirFunc  closedir,
        IOD_RewinddirFunc rewinddir,
        IOD_ReaddirFunc   readdir
    );
```

This is similar to the **IOD_install_driver()** call, but it includes additional functions that allow the program to perform file system-specific calls, mostly functions that manipulate directories in addition to files. Again, the functions **unlink**, **link**, **mkdir**, **rmdir**, **access**, **opendir**, **closedir**, **rewinddir** and **readdir** provide additional POSIX.1 system call functionality for the driver. The program can use these file system functions to create, remove, open, read and close directories. Note that TCS file system run-time support does not include a notion of current working directory.

The **examples** directory in the TriMedia Compilation System distribution includes several examples that demonstrate the use of file i/o functions. The examples/peripherals/hdvotest example installs a simple driver that recognizes only the file name “console.” Most of the functions in its **IOD_install_driver()** command are accordingly Null. examples/dynamic_loading/flash_demo gives a more interesting example demonstrating the installation of a file i/o driver for a flash file system.

The simple example below gives an example of a driver directing the output of a file using a circular memory buffer. The memory address and length of the buffer are encoded in the filename, which is formatted so that the driver's recognition function can recognize it.

```
#define BUFLEN 100

main(){
   char  filename[100];
   char  filebuffer[BUFLEN+1];
   FILE *f;
   int i;

   _MEM_Driver_init();

   printf("buffer= 0x%08x\n", filebuffer);

   sprintf(filename, "/dev/mem@0x%x#%d", &filebuffer, BUFLEN);

   filebuffer[BUFLEN]= 0;
   f= fopen(filename, "w");

   if( f==NULL ){
      printf("File could not be opened\n");
   }else{
      for( i=0; i<10; i++ ){
         printf("%s\n",filebuffer);
         fprintf(f, "** this is line number %d **\n",i);
      }
      printf( filebuffer );

      fclose(f);
   }
}
```

```
#define NAME_MASK "/dev/mem@%i#%i"

typedef struct {
   Char    *buffer;
   Integer  length;
   Char    *printpos;
} *MemFILE;

static Boolean RecogMEM ( String path ){
   Char    *buffer;
   Integer  length;
   Integer  result;

   result= sscanf( path, NAME_MASK, &buffer, &length );
   return result == 2;
}
static Integer OpenMEM(
   String    path,
   Integer   oflag,
   Integer   mode
){
   Integer  i;
   MemFILE  result= malloc(sizeof(*result));

   if (result == NULL) {
      return -1;
   }else{
      sscanf( path, NAME_MASK, &result->buffer, &result->length );
      result->printpos= result->buffer;
      for( i=0; i<result->length; i++ ){
         result->buffer[i]= '#';
      }
      return (Integer)result;
   }
}
static Integer WriteMEM(
   Integer   file,
   Char     *buf,
   Integer   nbyte )
){
   MemFILE  descr = (MemFILE)file;
   Char    *bound = descr->buffer + descr->length;
   Char    *printpos = descr->printpos;
   Integer  i;

   for (i=1; i<=nbyte; i++) {
      Char c= *(buf++);
      if (c == '\n') c= '@';
      *(printpos++) = c;
      if (printpos == bound) printpos= descr->buffer;
   }

    descr->printpos= printpos;
    return nbyte;
}
void_MEM_Driver_init(){
   IOD_install_driver( RecogMEM, NULL, NULL, OpenMEM, CloseMEM,
                       NULL, WriteMEM, NULL, IsattyMEM, StatMEM );
}
```

# Chapter 3

# Host Windows Interfaces

## TriMedia Manager Architectural Overview

The TriMedia Manager provides inter-processor communication functionality. It facilitates applications running on the host processor (x86) to communicate with applications running on the target processor (TriMedia).

The TriMedia Manager provides the following functionality:

- Downloading and executing TriMedia executables on the TriMedia processor.
- Message passing from host to target and vice versa.
- Event signaling from the host on the target and vice versa.
- Page-locking buffers on the host and enabling the target to access them.
- Allocation and freeing of shared memory (shared between host and target).
- An Application Programming Interface (API) for accessing the above functionality.

The TriMedia host driver currently runs on the following platforms:

- Windows 95
- Windows 98
- Windows NT 4.0
- Windows 2000 (Beta 3)
- Windows CE 2.1

The TriMedia manager consists of various components running on the host as well as the target. These components interface with each other to implement the TriMedia Manager functionality. Figure 5 shows the various components and the interactions between them.

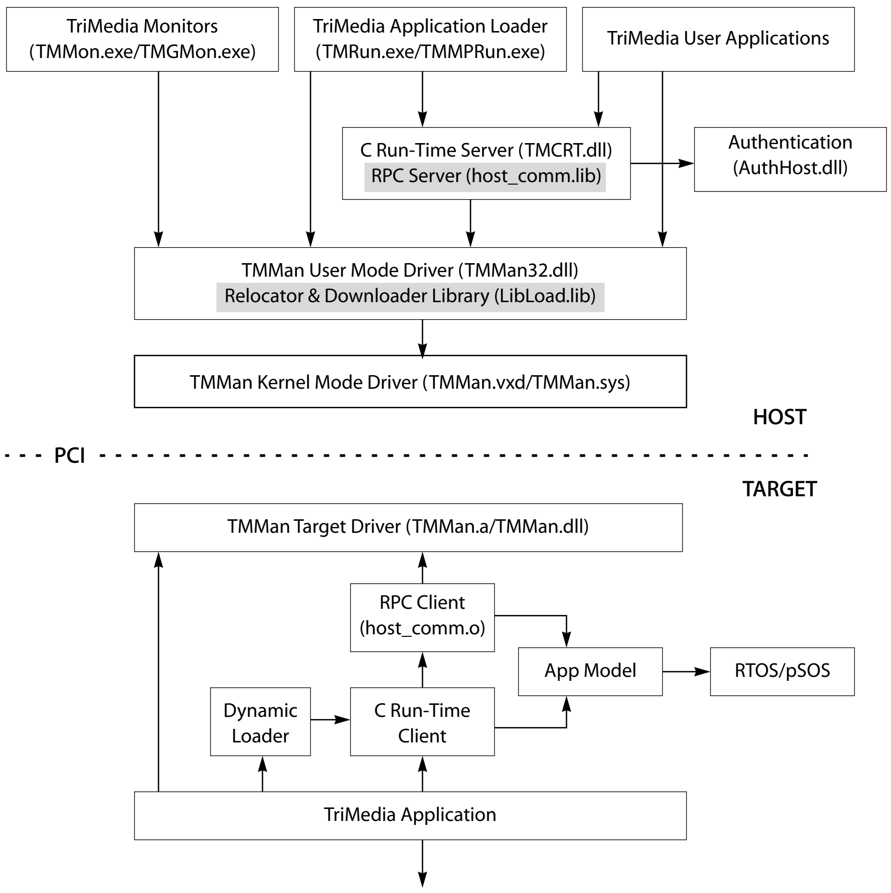
TriMedia Monitors
(TMMon.exe/TMGMon.exe)
TriMedia Application Loader
(TMRun.exe/TMMPRun.exe)
TriMedia User Applications
C Run-Time Server (TMCRT.dll)
RPC Server (host_comm.lib)
Authentication
(AuthHost.dll)
TMMan User Mode Driver (TMMan32.dll)
Relocator & Downloader Library (LibLoad.lib)
TMMan Kernel Mode Driver (TMMan.vxd/TMMan.sys)
HOST
PCI
TARGET
TMMan Target Driver (TMMan.a/TMMan.dll)
RPC Client
(host_comm.o)
App Model
RTOS/pSOS
Dynamic
Loader
C Run-Time
Client
TriMedia Application

**Figure 5** Host Interface Software Architecture

## Windows TMMan Modules

The following are the modules currently supported on the Windows version of TMMan:

| Module | Description |
|---|---|
| TMMan.sys | This is the kernel mode driver that provides bulk of the TMMan functionality. There are two versions of TMMan.sys: one for Windows 98/2000 and the other for Windows NT 4.0. |
| TMMan.vxd | This is a kernel mode driver that provides the bulk of TMMan functionality under Windows 95/98. |
| TMMan32.dll | User Mode Win32 DLL that provides the TMMan application programming interface to Win32 applications. This DLL simply calls TMMan.sys for the TMMan functionality. |
| TMRun.exe | Command line utility (Win32 console application) for downloading and running executables on the TriMedia processor. This program is also used by TMMon as the TriMedia console. |
| TMmpRun.exe | Multiprocessor version of TMRun, this enables multiprocessor cluster downloading on multiple TriMedia boards plugged in the system. |
| TMMon.exe | TriMedia Monitor - This is a interactive shell for downloading and running programs on TriMedia. It is a Win32 console mode application that provides a command based interface. TMMon reads its input and writes its output via standard handles so the input to TMMon including command can be redirected from an input file. |
| TMCRT.dll | TriMedia C Run Time server. This module accepts requests from the target and serves them. These requests are Unix level 2 I/O calls that are generated by the executable program running on TriMedia. It uses the TMMan messaging mechanism to communicate with the target. |
| TMMan.a | The target component of TMMan. This is a static library that boot applications on TriMedia are linked with. This module provides the TMMan functionality on the target. |
| Driver.exe | This is a helper utility that is provided in order to install the kernel mode driver in the system. This utility is required only at the time of installing the software and at the time of uninstalling. |
| TMManins.exe | Helper utility that automatically removes the TriMedia device and associated drivers from the system. After running this utility, restart the system to re-install the drivers. |
| TMGMon.exe | Graphical TriMedia Monitor. It’s a Win32 GUI application for downloading and executing TriMedia applications. |

# Chapter 4

# tmAvFormats.h: Multimedia Format Definitions

**Note**
For a general overview of TriMedia device libraries, see Chapter 5, *Device Libraries*.

## Audio Video Formats API Overview

The TriMedia Audio Video Formats Application Programming Interface (API) provides the common audio and video media formats used in the TriMedia Application Libraries components. The file tmAvFormats.h defines a set of formats which are used throughout the TriMedia system to identify multimedia data streams. The tmAvFormats.h provides only definitions. It does not contains any function prototypes.

### Definitions

The Library provides definitions of format types and data packets for certain types, as well as a definition for the data packet header.

#### Type Definitions

There are four types that are defined here:

- Enumerated types describing data.
- Format structure types.
- Packet structure types.
- Data Packets.

##### Enumerated Types

Components and data streams are classified using a number of fields. The values used for these fields are declared as enumerated types in tmAvFormats.h

##### Format Structures

The format of a data stream is given with a format structure. The generic format structure is the **tmAvFormat_t**. Audio and video data streams are each identified using their own type, but the audio and video format types are "subclassed" from the **tmAvFormat_t**. Effectively, this means that the start of the subclassed structures is identical to its parent structure, and the **tmAudioFormat_t** can be addressed like a **tmAvFormat_t**.

Formats can be used in two ways: A capabilities format will specify the (possibly many) formats acceptable to a component. A specification format requests a specific format.

##### Format Manager

The TSA default OL layer includes a format manager. This tool is used when connecting two components. The format manager will check whether a specified format is acceptable to the specification given in the capabilities structure. The format manager also checks packets for consistency when they are passed between components.

## Data Packets

In the TSA architecture, data of a specific type travels from one component to another component. The packet struct definitions make sure that the producer and the consumer of a certain type of data can pass information between each other without any problems. Each packet must contain as its first field a pointer to a standard packet header. Packets may contain any number of data buffers, although it is common for a packet to contain one data buffer.

## Audio Video Formats API Data Structure Descriptions

This section describes the Audio Video Formats API data structures.

| Name | Page |
|---|---|
| tmAvFormat_t | 35 |
| tmAudioFormat_t | 36 |
| tmVideoFormat_t | 38 |
| tmAvPacket_t | 40 |
| tmAvBufferDescriptor_t | 41 |
| tmAvHeader_t | 42 |
| tmComponentClass_t | 44 |
| tmAvDataClass_t | 44 |
| tmAvDataType_t | 45 |
| tmAvDataSubtype_t | 45 |
| tmSystemTypeFormat_t | 46 |
| tmVideoTypeFormat_t | 47 |
| tmAudioTypeFormat_t | 48 |
| tmControlTypeFormat_t | 49 |
| tmOtherTypeFormat_t | 49 |
| tmVideoRGBYUVFormat_t | 50 |
| tmVideoFlags_t | 51 |
| tmAudioPcmFormat_t | 53 |
| tmAudioMPEGFormat_t | 59 |
| tmVideoMPEGFormat_t | 59 |
| tmMPEG2TransportStreamFormat_t | 60 |
| tmVideoAnalogStandard_t | 60 |
| tmVideoAnalogAdapter_t | 61 |
| tmAudioAnalogAdapter_t | 61 |
| tmSSIAnalogConnection_t | 62 |
| tmTimeStamp_t | 62 |

## tmAvFormat_t

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

### Fields

| | |
|---|---|
| `size` | Size of this structure, i.e., sizeof(**tmAvFormat_t**). |
| `hash` | Reserved for system use. Initialize to zero. |
| `referenceCount` | Reserved for system use. Initialize to zero. |
| `dataClass` | One of the av format classes (Refer to **tmAvDataClass_t** on page 44). |
| `dataType` | One of the Audio/Video (av) types, dependent on **dataClass**. |
| `dataSubtype` | One of the av subtypes, dependent on **dataType**. |
| `description` | Optional extension description of a format (format-specific). |

### Description

The **tmAvFormat_t** is a fundamental building block of the TriMedia Software Streaming Architecture. It is used to describe a class, a type, a subtype and an optional description. **dataType** and **subType** are used to describe the stream.

The exact contents depend on the **dataClass**. For instance, if **dataClass** is **avdcAudio**, then **dataType** might be **aftLinearPCM** and **dataSubtype** might be **apfStereo16**.

The description field allows a particular module to implement more (unspecified) information. Some data formats will be defined as subclasses of this basic format. For example, the audio format (**tmAudioFormat_t**) also defines a field for the sample rate of the data stream.

## tmAudioFormat_t

```
typedef struct tmAudioFormat_t{
   UInt16            size;
   UInt16            hash;
   UInt32            referenceCount;
   tmAvDataClass_t   dataClass;
   UInt32            dataType;
   UInt32            dataSubtype;
   UInt32            description;
   float             sampleRate;
} tmAudioFormat_t, *ptmAudioFormat_t;
```

### Fields

| | |
|---|---|
| `size` | Size of this structure, i.e., sizeof(**tmAudioFormat_t**). |
| `hash` | Reserved for system use. Initialize to zero. |
| `referenceCount` | Reserved for system use. Initialize to zero. |
| `dataClass` | One of the av format classes (see **tmAvDataClass_t**). |
| `dataType` | One of the av types, dependent on **dataClass**. |
| `dataSubtype` | One of the av subtypes, dependent on **dataType**. |
| `description` | Optional extension description of a format (format-specific). This field serves the following two purposes: |

(1) Specifies whether the left and right channels carry Dolby Surround encoded data. The bitmask **AVFORMAT_PROLOGIC_ENCODED** can be used to access this information as in the following example:

```
tmAudioFormat_t stereo16;
if( stereo16.description & AVFORMAT_PROLOGIC_ENCODED)
   printf(" audio data is Surround encoded\n");
```

(2) If **dataSubtype** represents a 32-bit type, specifies how many bits are significant. This information is stored in the last 8 bits of the description field and can be accessed with the bitmask **AVFORMAT_NUMBER_OF_BITS_MASK**.

```
tmAudioFormat_t stereo32;
Int             noOfBits;
noOfBits = stereo32.description &
           AVFORMAT_NUMBER_OF_BITS_MASK;
```

| | |
|---|---|
| `sampleRate` | The sample rate for this audio format, in Hertz. |

### Description

This struct is a subtype of **tmAvFormat_t**. The initial fields are identical. The audio-specific field, **sampleRate** is added.

## Implementation Notes

Example of a linear 16-bit stereo audio:

```
tmAudioFormat_t audio16stereo = {
   sizeof(tmAudioFormat_t),   // size
   0,                         // hash
   0                          // referenceCount
   avdcAudio,                 // dataClass
   atfLinearPCM,              // dataType
   apfStereo16                // dataSubtype
   0,                         // description
   44100.0                    // sampleRate
};
```

Example of an undecoded AC3 stream:

```
tmAudioFormat_t ac3audio = {
   sizeof(tmAudioFormat_t),   // size
   0,                         // hash
   0                          // referenceCount
   avdcAudio,                 // dataClass
   atfAC3,                    // dataType
   apfNone,                   // dataSubtype
   0,                         // description
   48000                      // sampleRate
};
```

Example of a linear 20-bit stereo audio:

```
tmAudioFormat_t audio32stereo = {
   sizeof(tmAudioFormat_t),   // size
   0,                         // hash
   0                          // referenceCount
   avdcAudio,                 // dataClass
   atfLinearPCM,              // dataType
   apfStereo32                // dataSubtype
   20,                        // description: number of bits
   44100.0                    // sampleRate
};
```

Data of this type is stored in 32-bit aligned words.

## tmVideoFormat_t

```
typedef struct tmVideoFormat_t {
   UInt16                  size;
   UInt16                  hash;
   UInt32                  referenceCount;
   tmAvDataClass_t         dataClass;
   UInt32                  dataType;
   UInt32                  dataSubtype;
   UInt32                  description;
   UInt32                  imageWidth;
   UInt32                  imageHeight;
   UInt32                  imageStride;
   UInt32                  activeVideoStartX;
   UInt32                  activeVideoStartY;
   UInt32                  activeVideoEndX;
   UInt32                  activeVideoEndY;
   tmVideoAnalogStandard_t videoStandard;
} tmVideoFormat_t, *ptmVideoFormat_t;
```

### Fields

| | |
|---|---|
| `size` | Size of this structure, i.e., sizeof(tmVideoFormat_t). |
| `hash` | Reserved for system use. Initialize to zero. |
| `referenceCount` | Reserved for system use. Initialize to zero. |
| `dataClass` | One of the av format classes (Refer to **tmAvDataClass_t** on page 44). |
| `dataType` | One of the av types, dependent on **dataClass**. |
| `dataSubtype` | One of the av subtypes, dependent on **dataType**. |
| `description` | Optional extension description of a format (format specific). |
| `imageWidth` | Horizontal image size (in pixels). |
| `imageHeight` | Vertical image size (in lines). |
| `imageStride` | Distance in bytes from beginning of one line of video to the next line of video. |
| `activeVideoStartX` | Starting X position of active video. Active video means the part of the video that is to be displayed. It could contain section of VBI. |
| `activeVideoStartY` | Starting Y position of active video. Together with **activeVideoStartX**, it defines the starting point in image domain given by (X, Y). |
| `activeVideoEndX` | End X position of active video. |

| | |
|---|---|
| `activeVideoEndY` | End Y position of active video. Together with `activeVideoStartX`, it defines the ending point in image domain given by (X, Y). |
| `videoStandard` | Defines the analog video standard to be used. For details of supported analog video standard please see the description of **tmVideoAnalogStandard_t**. |

## Description

This struct is a subtype of **tmAvFormat_t**.

## Implementation Notes

Example of MPEG-1 system stream:

```
tmVideoFormat_t mpeg1system = {
   sizeof(tmVideoFormat_t),   // size
   0,                         // hash
   0,                         // referenceCount
   avdcSystem,                // dataClass
   stfMPEG1System,            // dataType
   avdsNone,                  // dataSubtype
   0,                         // description
   320,                       // hsize
   240,                       // vsize
   0,                         // activeVideoStartX
   0,                         // activeVideoStartY
   320,                       // activeVideoEndX
   240,                       // activeVideoEndY
   vasNTSC,                   // videoStandard
};
```

Example of NTSC stream:

```
tmVideoFormat_t ntscVideo = {
   sizeof(tmVideoFormat_t),             // size
   0,                                   // hash
   0,                                   // referenceCount
   avdcVideo                            // dataClass
   vtfYUV                               // dataType
   vdfYUV422Planar                      // dataSubtype
   vdfInterlaced | vdfFrameRate_29_97,  // description
   720,                                 // hsize
   480,                                 // vsize
   0,                                   // activeVideoStartX
   0,                                   // activeVideoStartY
   720,                                 // activeVideoEndX
   480,                                 // activeVideoEndY
   vasNTSC,                             // videoStandard
};
```

## tmAvPacket_t

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

### Fields

| | |
|---|---|
| `header` | Pointer to the packet header (see **tmAvHeader_t** on page 42). Since the TSSA default functions use it, all legal packets must contain a valid header. |
| `allocatedBuffers` | How many buffers are allocated for the packet. Must be set by the application that creates and owns the packet. Remains constant over the life of the packet. |
| `buffersInUse` | This field, along with the **dataSize** fields of the attached buffers (see **tmAvBufferDescriptor_t** on page 41), were designed to accurately indicate the state of a packet.<br><br>If the packet is full, **buffersInUse** == **allocatedBuffers** and **dataSize** == **bufSize** for each buffer.<br><br>If the packet is empty, **buffersInUse == 0** and **dataSize == 0** for each buffer.<br><br>Partially full states in between these two extremes are allowed, and their interpretation is up to the components involved. This mechanism is implemented in the TSA default functions.<br><br>**buffersInUse** must be set accurately by each component before it sends the packet to dataout (**putfull**). Packets with zero **buffersInUse** are treated as empty by the default dataout function: They are shunted to the empty queue. It is customary for a component to set **buffersInUse** to zero before putting the packet to the empty queue. The default datain function does not examine this field. |
| `buffers` | An array of buffer descriptors. See **tmAvBufferDescriptor_t** on page 41. |

### Description

**tmAvPacket_t** structs define the fundamental unit of data passed around in the TriMedia Software Architecture. 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. For further information about packet infrastructure, see Chapter 7, *TSSA Essentials*.

## tmAvBufferDescriptor_t

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

### Fields

| | |
|---|---|
| `bufSize` | Indicates the amount of memory allocated for that buffer. Must be set by the application that creates and owns the packet. |
| `dataSize` | In your component, you must set **dataSize** in each **tmAvBufferDescriptor_t** when you fill the buffer. The size of data in a given buffer can be less than or equal to **bufsize**. The default functions do not examine **datasize**, so the interpretation of a value of zero is up to the component. |

### Description

Works in conjunction with **tmAvFormat_t** (see page 35).

## tmAvHeader_t

```
typedef struct tmAvHeader_t {
   UInt32          id;
   UInt32          flags;
   Pointer         userSender;
   Pointer         userReceiver;
   Pointer         userPointer;
   tmTimeStamp_t   time;
   Pointer         format;
} tmAvHeader_t, *ptmAvHeader_t;
```

### Fields

| | |
|---|---|
| `id` | Used to identify packets while debugging. For the use of the application. Not used internally by TSSA components. |
| `flags` | A few specific values are recognized by TSSA components. One such value is the **timestampvalid** flag. Another value is **avhField2**. It specifies whether the packet contains Field 1 or Field 2, when the Video Renderer is used on a field basis. |
| `userSender`, `userReceiver` | These two fields are not interpreted by the defaults and they can be used as a programmer sees fit. They are declared as pointers, but the data structures to which they point are not preserved by generic operations (such as the copying of packets from one processor to another). Two fields are provided so that a distinction can easily be made between data written by the sender and data written by the receiver. |
| `userPointer` | This field gives users a way to attach more application-specific data to the header of a packet. Three basic rules govern the use of this field. These rules allow user data to be preserved through generic operations, such as the copying of packets from one processor to another. (When considering the use of this field, remember that another option would be to attach more buffers to the packet.)<br><br>The memory to which these fields point:<br>- must be a data structure.<br>- must have a 16-bit size as its first field.<br>- must not have any further pointers. |
| `time` | Time stamp (can be used, for example, for presentation time). |

| | |
|---|---|
| `format` | Subclass of **tmAvFormat_t**. If Null, the component is directed to reuse the previously specified format. |

## Description

Works in conjunction with **tmAvFormat_t** (see page 35).

## tmComponentClass_t

```
typedef enum {
   ccClock,
   ccSystemDecoder,
   ccVideoDecoder,
   ccVideoEncoder,
   ccVideoRenderer,
   ccVideoDigitizer,
   ccVideoTransform,
   ccAudioDecoder,
   ccAudioEncoder,
   ccAudioRenderer,
   ccAudioDigitizer,
   ccAudioTransform,
   ccGraphics2D,
   ccGenericIn,
   ccGenericOut
} tmComponentClass_t;
```

### Description

The **tmComponentClass_t** is used during the negotiation process as two components attempt to connect. The component class defines what type of data processing a TriMedia application library component does, for example, video renderer. The component class is returned as one of the default values of the **GetCapabilities** call on a component. Generic components do not interpret the data that they operate on, regardless of the data type: for example, audio or video. The fields in this enum are mutually exclusive.

## tmAvDataClass_t

```
typedef enum {
   avdcGeneric = 0xffffffff,
   avdcSystem =  0x00000001,
   avdcVideo =   0x00000002,
   avdcAudio =   0x00000004,
   avdcControl = 0x00000008,
} tmAvDataClass_t;
```

### Description

The **tmAvDataClass_t** is used during the negotiation process as two components attempt to connect. The data class enum type defines the data classes for TriMedia. One of these enums will be used in **dataClass** field of **tmAvFormat_t**.

## tmAvDataType_t

```
typedef enum {
   avdtGeneric   = 0xffffffff
   avdtNone      = 0x80000000
} tmAvDataType_t;
```

### Description

Used to describe or specify data types when no particular type is appropriate. Generic means that any type is accepted by a capabilities format. None is used when no data type is required. This is stated explicitly.

## tmAvDataSubtype_t

```
typedef enum {
   avdsGeneric   = 0xffffffff
   avdsNone      = 0x80000000
} tmAvDataSubtype_t;
```

### Description

Used to describe or specify data subtypes when no particular subtype is appropriate. Generic means that any subtype is accepted by a capabilities format. None is used when no data subtype is required. This is stated explicitly.

## tmSystemTypeFormat_t

```
typedef enum {
   stfGeneric          = 0xffffffff,
   stfNone             = 0x80000000,
   stfMPEG1System      = 0x00000001,
   stfMPEG2Program     = 0x00000002,
   stfMPEG2Transport   = 0x00000004,
   stfQuicktimeMovie   = 0x00000008,
   stfDVD              = 0x00000010
} tmSystemTypeFormat_t;
```

### Description

The **tmSystemTypeFormat_t** is used by components which support MPEG system streams (avdcSystem). This enum defines the data type belonging to each data class. It can be used in the **dataType** field of **tmAvFormat_t**. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

## tmVideoTypeFormat_t

```
typedef enum {
   vtfGeneric          = 0xffffffff,
   vtfNone             = 0x80000000,
   vtfReserved1        = 0x00000001,
   vtfReserved2        = 0x00000002,
   vtfQuicktimeVideo   = 0x00000004,
   vtfYUV              = 0x00000010,
   vtfRGB              = 0x00000020,
   vtfJPEG             = 0x00000040,
   vtfMPEG             = 0x00000080,
   vtfDVC              = 0x00000100,
   vtfH261             = 0x00000200,
   vtfH263             = 0x00000400,
} tmVideoTypeFormat_t;
```

### Description

The **tmVideoTypeFormat_t** is used by components which support video streams (avdcVideo). This enum defines the data type belonging to each data class. It can be used in the **dataType** field of **tmAvFormat_t**. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

### Implementation Notes

**vtfMPEG1Video** and **vtfMPEG2Video** are obsolete. Use **vtfMPEG** and **subtype**.

## tmAudioTypeFormat_t

```
typedef enum {
   atfGeneric          = 0xffffffff,
   atfNone             = 0x80000000,
   atfMuLaw            = 0x00000001,
   atfALaw             = 0x00000002,
   atfADPCM            = 0x00000004,
   atfMPEG             = 0x00000008,
   atfG723             = 0x00000010,
   atfG728             = 0x00000020,
   atfG729             = 0x00000040,
   atfQuicktimeAudio   = 0x00000080,
   atfReserved1        = 0x00000100,
   atfReserved2        = 0x00000200,
   atfReserved3        = 0x00000400,
   atfSDDS             = 0x00000800,
   atfAC3              = 0x00001000,
   atfDTS              = 0x00002000,
   atfLinearPCM        = 0x00004000,
   atf1937             = 0x00008000,
} tmAudioTypeFormat_t;
```

### Description

The **tmAudioTypeFormat** is used by components which support audio streams (**avdcAudio**). Audio subtypes may be further specified. This enum describes the formats of audio streams. Some streams (commonly **atfLinearPCM**) are more completely described with an additional **subtype** field. It can be used in the **dataType** field of **tmAvFormat_t**. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

### Implementation Notes

**atfMPEG1Audio**, **atfMPEG2Audio**, and **atfMPEG2AugmentedAudio** are obsolete. Use **atfMPEG** and **subtype**.

## tmControlTypeFormat_t

```
typedef enum {
   ctfGeneric   = 0xffffffff,
   ctfNone      = 0x80000000
} tmControlTypeFormat_t;
```

### Description

This enum defines the data type belonging to each data class. It can be used in the **dataType** field of **tmAvFormat_t**.

## tmOtherTypeFormat_t

```
typedef enum {
   otfGeneric   = 0xffffffff,
   otfNone      = 0x80000000
} tmOtherTypeFormat_t;
```

### Description

This enum defines the Audio, Video, etc. format subtypes. It can be used in the **dataType** field of **tmAvFormat_t**.

## tmVideoRGBYUVFormat_t

```
typedef enum {
   vdfGeneric                = 0xffffffff,
   vdfNone                   = 0x80000000,
   vdfRGB8A_233              = 0x00000001,
   vdfRGB8R_332              = 0x00000002,
   vdfRGB15Alpha             = 0x00000004,
   vdfRGB16                  = 0x00000008,
   vdfRGB24                  = 0x00000010,
   vdfRGB24Alpha             = 0x00000020,
   vdfYUV420Planar           = 0x00000040,
   vdfYUV422Planar           = 0x00000080,
   vdfYUV411Planar           = 0x00000100,
   vdfYUV420Interspersed     = 0x00000200,
   vdfYUV422Interspersed     = 0x00000400,
   vdfYUV411Interspersed     = 0x00000800,
   vdfYUV422Sequence         = 0x00001000,
   vdfYUV422SequenceAlpha    = 0x00002000,
   vdfMono                   = 0x00004000,
   vdfYUV444Planar           = 0x00008000,
   vdfDTVCMPlanar            = 0x00010000, /* for use in DTV */
   vdfDTVCMSequence          = 0x00020000, /* for use in DTV */
   vdfYInterleavedUV420      = 0x00040000,
   vdfYUVPlanarAlpha4        = 0x00080000
} tmVideoRGBYUVFormat_t;
```

### Description

RGB or YUV type data streams further identify their format using these subtypes. This enum defines the data subtypes belonging to the **vtfRGBYUV** video type format (It lists possible formats of video data). It can be used in the **dataSubtype** field of **tmAvFormat_t**. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

**vdfDTVCMPlanar** and **vdfDTVCMSequence** are for use in the DTV platform.

**Note**
DTVCM stands for Digital TV Color Multiplexer.

### Implementation Notes

**vdfYUV422Sequence**: U0, Y0, V0, and Y1 are for VO overlay. **vdfYUV422SequenceAlpha**: U0, Y0, V0, and Y1 are for VO overlay, with low bit for alpha blending. **vdfMono**: 8-bit monochrome.

## tmVideoFlags_t

```
typedef enum {
   vdfInterlaced          = 0x1,
   vdfBottomFieldFirst    = 0x2,
   vdfFieldInFrame        = 0x4,
   vdfFieldInField        = 0x8,
   vdfProgressive         = 0x10,
   vdfFrameRate_23_976    = 0x100,
   vdfFrameRate_24        = 0x200,
   vdfFrameRate_25        = 0x400,
   vdfFrameRate_29_97     = 0x800,
   vdfFrameRate_30        = 0x1000,
   vdfFrameRate_50        = 0x2000,
   vdfFrameRate_59_94     = 0x4000,
   vdfFrameRate_60        = 0x8000,
   vdfDTVCM_0Video        = 0x10000,  /* for use in DTV */
   vdfDTVCM_25Video       = 0x20000,  /* for use in DTV */
   vdfDTVCM_50Video       = 0x40000,  /* for use in DTV */
   vdfDTVCM_75Video       = 0x80000,  /* for use in DTV */
   vdfDTVCM_100Video      = 0x100000, /* for use in DTV */
   vdfMPEGExtension       = 0x20000,
   vdfAspectRatio_4x3     = 0x100000,
   vdfAspectRatio_16x9    = 0x200000,
} tmVideoFlags_t;
```

### Description

The video flags are used in the description field of the **tmVideoFormat_t**. They are used to give more information about the video stream.

| | |
|---|---|
| `vdfInterlaced` | This bit is set if the video stream is meant for an NTSC or PAL type of interlaced display. |
| `vdfBottomFieldFirst` | This bit is set if the field order is not the normally expected "top field first." MPEG2 allows this type of stream. |
| `vdfFieldInField` | Meant for interlaced displays. |
| `vdfFieldInFrame` | Meant for interlaced displays. |
| `vdfProgressive` | Meant for progressive displays. |
| `vdfFrameRate` | The frame rate bits are used by MPEG encoders. |
| `vdfDTVCM_0Video` | Blends 0% video and 100% of graphics. |
| `vdfDTVCM_25Video` | Blends 25% video and 75% of graphics. |
| `vdfDTVCM_50Video` | Blends 50% video and 50% of graphics. |
| `vdfDTVCM_75Video` | Blends 75% video and 25% of graphics. |
| `vdfDTVCM_100Video` | Blends 100% video and 0% of graphics. |

| | |
|---|---|
| `vdfAspectRatio_4x3` | Use 4x3 aspect ratio. |
| `vdfAspectRatio_16x9` | Use 16x9 aspect ratio. |

These fields are restricted to the low 16 bits of the description field. Use **vdfFieldInField** and **vdfFieldInFrame** to indicate that video packets contain one field only. Use **vdfInterlaced** and **vdfProgressive** to indicate that video packets contain one entire frame. Figure 6 describes the contents of the buffer parts of the different video packets.

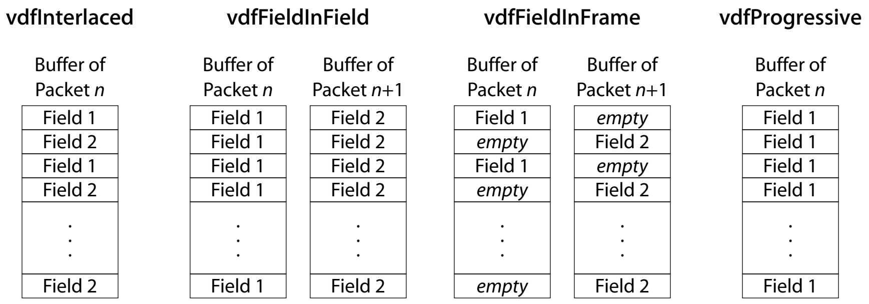

**Figure 6** Contents of Video Packet Buffers

## tmAudioPcmFormat_t

```
typedef enum {
   apfGeneric            = 0xffffffff,
   apfNone               = 0x80000000,
   apfMono8              = 0x00000001,
   apfStereo8            = 0x00000002,
   apfMono16             = 0x00000004,
   apfStereo16           = 0x00000008,
   apfFourCh_3_1_0_16    = 0x00000010,
   apfFourCh_2_2_0_16    = 0x00000020,
   apfFourCh_2_1_1_16    = 0x00000040,
   apfFourCh_3_0_1_16    = 0x00000080,
   apfFiveDotOne16       = 0x00000100,
   apfSevenDotOne16      = 0x00000200,
   apfMulti16            = 0x00000400,
   apfMono32             = 0x00000800,
   apfStereo32           = 0x00001000,
   apfFourCh_3_1_0_32    = 0x00002000,
   apfFourCh_2_2_0_32    = 0x00004000,
   apfFourCh_2_1_1_32    = 0x00008000,
   apfFourCh_3_0_1_32    = 0x00010000,
   apfFiveDotOne32       = 0x00020000,
   apfSevenDotOne32      = 0x00040000,
   apfMulti32            = 0x00080000,
   apfMonoFloat          = 0x00100000,
   apfStereoFloat        = 0x00200000,
   apfFourCh_3_1_0_float = 0x00400000,
   apfFourCh_2_2_0_float = 0x00800000,
   apfFourCh_2_1_1_float = 0x01000000,
   apfFourCh_3_0_1_float = 0x02000000,
   apfFiveDotOneFloat    = 0x04000000,
   apfSevenDotOneFloat   = 0x08000000,
   apfMultiFloat         = 0x10000000,
   apfReserved           = 0x20000000,
   apfReserved2          = 0x40000000,
} tmAudioPcmFormat_t;
```

### Description

Streams of linear PCM audio data with the type **atfLinearPCM** further identify their format using these subtypes. To identify the format of a stream, only one value can be used. To identify the capabilities of a component, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

The following table illustrates the order of samples in memory for the different PCM audio subtypes. The channel abbreviations mean:

| Abbreviation | Channel |
|---|---|
| L | Left |
| R | Right |
| C | Center |
| Ls | Left Surround |
| Rs | Right Surround |
| S | Surround (but if only one Surround is present) |
| Lc | Left Center |
| Rc | Right Center |
| Lfe | Low Frequency Effects |

The audio data is in packed interleaved format. With 16-bit data, this is very efficient. When 18 or 20-bit data is used, it is stored in 32-bit format.

When the 32-bit formats are used, the specific description field of the **tmAudioFormat_t** which is used is filled in with the number of bits which are significant (for example, 18, 20, etc.). This lets components know how to clip or scale their final results.

**Note**
TM1xxx hardware does not support 32-bit data in a native fashion. Some systems provide external hardware to support 32-bit data. The TM DTV hardware uses the lower 20 bits of 32. It also inserts some sync bits into the unused high bits of the first word.

**apfMono8**

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| C,C,C,C | C,C,C,C | C,C,C,C | C,C,C,C | C,C,C,C | C,C,C,C | C,C,C,C | C,C,C,C |

**apfStereo8**

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L,R,L,R | L,R,L,R | L,R,L,R | L,R,L,R | L,R,L,R | L,R,L,R | L,R,L,R | L,R,L,R |

**apfMono16**

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| C, C | C, C | C, C | C, C | C, C | C, C | C, C | C, C |

**apfStereo16**

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L, R | L, R | L, R | L, R | L, R | L, R | L, R | L, R |

**apfFourCH_3_1_0_16** Three front (Left, Right, Center) one surround, and no subwoofer.

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L, R | C, S | L, R | C, S | L, R | C, S | L, R | C, S |

**apfFourCH_2_2_0_16** Two front (Left, Right) two surround (Ls and Rs), and no subwoofer.

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L, R | Ls, Rs | L, R | Ls, Rs | L, R | Ls, Rs | L, R | Ls, Rs |

**apfFourCH_2_1_1_16** Two front (Left, Right), one surround (S), and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L, R | S, Lfe | L, R | S, Lfe | L, R | S, Lfe | L, R | S, Lfe |

**apfFourCH_3_0_1_16** Three front (Left, Right, Center) no surround, and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L, R | C, Lfe | L, R | C, Lfe | L, R | C, Lfe | L, R | C, Lfe |

**apfFiveDotOne16** Three front (Left, Right, Center,) two surround (Ls, Rs), and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L, R | C, Lfe | Ls, Rs | L, R | C, Lfe | Ls, Rs | L, R | C, Lfe |

**apfSevenDotOne16** Three front (Left, Right, Center) four surround (Ls, Rs, Lc, and Rc), and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L, R | C, Lfe | Ls, Rs | Lc, Rc | L, R | C, Lfe | Ls, Rs | Lc, Rc |

apfMulti16

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| not specified | not specified | not specified | not specified | not specified | not specified | not specified | not specified |

apfMono32

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| C | C | C | C | C | C | C | C |

apfStereo32

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | L | R | L | R | L | R |

**apfFourCH_3_1_0_32** Three front (Left, Right, Center) one surround, and no subwoofer.

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | C | S | L | R | C | S |

**apfFourCH_2_2_0_32** Two front (Left, Right), two surround (Ls and Rs), and no subwoofer.

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | Ls | Rs | L | R | Ls | Rs |

**apfFourCH_2_1_1_32** Two front (Left, Right), one surround (S), and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | S | Lfe | L | R | S | Lfe |

**apfFourCH_3_0_1_32** Three front (Left, Right, Center,) no surround, and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | C | Lfe | L | R | C | Lfe |

**apfFiveDotOne32** Three front (Left, Right, Center) two surround (Ls, Rs), and one sub (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | C | Lfe | Ls | Rs | L | R |

**apfSevenDotOne32** Three front (Left, Right, Center) four surround (Ls, Rs, Lc, and Rc), and one sub (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | C | Lfe | Ls | Rs | Lc | Rc |

**apfMulti32**

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| not specified | not specified | not specified | not specified | not specified | not specified | not specified | not specified |

**apfMonoFloat**

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| C | C | C | C | C | C | C | C |

**apfStereoFloat**

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | L | R | L | R | L | R |

**apfFourCH_3_1_0_float** Three front (Left, Right, Center) one surround, and no sub-woofer.

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | C | S | L | R | C | S |

**apfFourCH_2_2_0_float** Two front (Left, Right), two surround (Ls and Rs), and no sub-woofer.

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | Ls | Rs | L | R | Ls | Rs |

**apfFourCH_2_1_1_float** Two front (Left, Right), one surround (S), and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | S | Lfe | L | R | S | Lfe |

**apfFourCH_3_0_1_float** Three front (Left, Right, Center) no surround, and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | C | Lfe | L | R | C | Lfe |

**apfFiveDotOneFloat** Three front (Left, Right, Center) two surround (Ls, Rs), and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | C | Lfe | Ls | Rs | L | R |

**apfSevenDotOneFloat** Three front (Left, Right, Center,) four surround (Ls, Rs, Lc, and Rc), and subwoofer (Lfe).

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| L | R | C | Lfe | Ls | Rs | Lc | Rc |

**apfMultiFloat**

| 32-bit Word 1 | 32-bit Word 2 | 32-bit Word 3 | 32-bit Word 4 | 32-bit Word 5 | 32-bit Word 6 | 32-bit Word 7 | 32-bit Word 8 |
|---|---|---|---|---|---|---|---|
| not specified | not specified | not specified | not specified | not specified | not specified | not specified | not specified |

## tmAudioMPEGFormat_t

```
typedef enum {
   amfGeneric          = 0xffffffff,
   amfNone             = 0x80000000,
   amfMPEG1_Layer1     = 0x00000001,
   amfMPEG1_Layer2     = 0x00000002,
   amfMPEG1_Layer3     = 0x00000004,
   amfMPEG2            = 0x00000008,
   amfMPEG1_Augmented  = 0x00000010,
   amfMPEG_AAC         = 0x00000020,
} tmAudioMPEGFormat_t;
```

### Description

This enum defines subtypes to identify MPEG audio formats. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

## tmVideoMPEGFormat_t

```
typedef enum {
   vmfGeneric   = 0xffffffff,
   vmfNone      = 0x80000000,
   vmfMPEG1     = 0x00000001,
   vmfMPEG2     = 0x00000002,
   vmfMPEG4     = 0x00000004,
} tmVideoMPEGFormat_t;
```

### Description

This enum defines subtypes to identify MPEG video formats. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

## tmMPEG2TransportStreamFormat_t

```
typedef enum {
   tsfGeneric         = 0xffffffff,
   tsfNone            = 0x80000000,
   tsfStandard        = 0x00000001,
   tsfTM2TimeStamped  = 0x00000002
} tmMPEG2TransportStreamFormat_t;
```

### Description

This enum defines subtypes to identify the MPEG2 transport stream System Type Format.

## tmVideoAnalogStandard_t

```
typedef enum {
   vasGeneric     = 0xffffffff,
   vasNone        = 0x80000000,
   vasNTSC        = 0x00000001,
   vasPAL         = 0x00000002,
   vasSECAM       = 0x00000004
   vas720x480p    = 0x00000010,  /* for use in DTV */
   vas768x576p    = 0x00000020,  /* for use in DTV */
   vas960x540p    = 0x00000080,  /* for use in DTV */
   vas1920x1080   = 0x00000100
} tmVideoAnalogStandard_t;
```

### Description

This enum defines possible standards for an analog video signal, for example, NTSC or PAL. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

| | |
|---|---|
| `vas720x480p` | Configures video output as 720×480 progressive display. |
| `vas768x576p` | Configures video output as 768×576 progressive display. |
| `vas960x540p` | Configures video output as 960×540 progressive display. |
| `vas1920x1080` | Configure video output as 1920×1080 interlaced. |

## tmVideoAnalogAdapter_t

```
typedef enum {
   vaaGeneric  = 0xffffffff,
   vaaNone     = 0x80000000,
   vaaCVBS     = 0x00000001,
   vaaSvideo   = 0x00000002,
   vaaExt1     = 0x00000004
} tmVideoAnalogAdapter_t;
```

### Description

This enum defines possible adapters for an analog video signal, for example, S-Video or composite. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

## tmAudioAnalogAdapter_t

```
typedef enum {
   aaaGeneric         = 0xffffffff,
   aaaNone            = 0x80000000,
   aaaMicInput        = 0x00000001,
   aaaLineInput       = 0x00000002,
   aaaAuxInput1       = 0x00000004,
   aaaAuxInput2       = 0x00000008,
   aaaDigitalInput    = 0x00000010,
   aaaLineOutput1     = 0x00000100,
   aaaLineOutput2     = 0x00000200,
   aaaAuxOutput1      = 0x00000400,
   aaaAuxOutput2      = 0x00000800,
   aaaDigitalOutput   = 0x00001000
} tmAudioAnalogAdapter_t;
```

### Description

This enum defines possible adapters for an analog audio signal, for example, aux or line. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

## tmSSIAnalogConnection_t

```
typedef enum {
   sacNoConnection      = 0x00,
   sacConnectToPOTS     = 0x01,
   sacConnectToISDN     = 0x02,
   sacConnectToOTHER    = 0x80
} tmSSIAnalogConnection_t;
```

### Description

Defines possible analog back ends for the TriMedia SSI peripheral, for example, ISDN or POTS. When the format is specified in a capabilities structure, a number of types may be OR'd together. This would indicate that any of the OR'd types are supported.

## tmTimeStamp_t

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

### Fields

| | |
|---|---|
| `ticks` | Low 32-bit clock tick counter (value interpreted based upon **clockType**). |
| `hiTicks` | High 32-bit clock tick counter (value interpreted based upon **clockType**). |

### Description

The time stamp field contains a 64-bit time value. Its interpretation is up to the clock that is selected when a component is setup. Clocks can be of various types, such as the 90 kHz clock used by MPEG, or SMPTE clocks. The clock handle passed in to a module's setup structure contains callback functions to access the clock values.

**Note**
For information on clock support, see Chapter 4, *Clock Support API*, of Book 5, *System Utilities*, Part A.

# Chapter 5

# Device Libraries

## Introduction

The TriMedia software architecture includes a layer that provides a public interface to hardware peripherals. This layer is generally known as the "device library" layer. A device library exports two interfaces. At the top, a device library gives an interface that is designed to be constant regardless of the current platform. At the bottom, a device library exports an interface that allows individual platforms to provide code appropriate to implement the public device library interface.

Device libraries are "operating system agnostic." They are not device drivers. They do not specify the method of data transfer. Device libraries allow applications to install this, either as a callback function, or as the entire interrupt service routine. An individual device library may contain more or less functionality. At the least, it provides a way to claim the resource associated with the library. More complicated device libraries include significant code to export a reasonable interface to a complicated device.

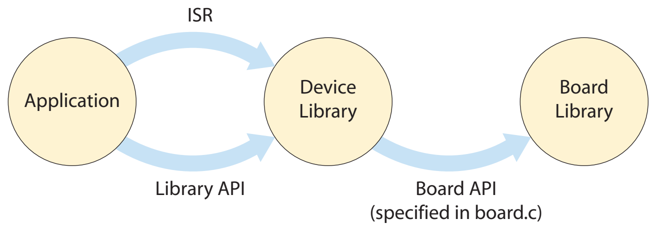

**Figure 7** Device Library Architecture

**Note**
Refer to Books 5–9 for information specific to a given device library. Refer to the appropriate TriMedia data book for details about TriMedia hardware.

Historically, device libraries were developed to support TriMedia's on-chip peripherals. But as the TriMedia software stack matured, the device library concept matured. Today, the TriMedia Software Architecture provides services that let users support any sort of hardware related device with an appropriate library.

The TriMedia toolset includes device library APIs supporting the peripherals that are found on all existing TM variants. Examples of these are the audio and video I/O devices. The SDK also offers what are effectively device library APIs to several components that are commonly found on boards connected to TriMedia through some sort of generic bus (PCI, IIC). The tsaUART API is an example of one of these.

## Naming Conventions

Public functions in device libraries are generally prefixed by tsaDev, where "Dev" is an abbreviation of the device name. Some older libraries may not follow this convention, but all new libraries do. The tsa prefix identifies the library as participating in the TriMedia Software Architecture.

Each device exports at least one **#include** file. A user should expect to include only one such file per library. If a library is built of several pieces, the include file for the “compound” library will incorporate the headers for its constituent parts.

Many device libraries also export a header file describing their MMIO (Memory Mapped I/O) interface. They do so using macros that follow the naming conventions from the appropriate TriMedia Data Book (for example, **aiBUF2_FULL**). There are macros for field identification and masking, which usually are not needed in user code. The device libraries and board support packages make use of this macro-based interface.

**Note**
In the following sections, the term “dev” stands for any device name. For example, in the type **devCapabilities_t**, “dev” can be replaced by ai, ao, vi, vo, vld, iic, mpeg, hdvo, evo, ssi, dma, int, pin, sem, exc, or tim. The exception to this rule is the name of the error code type **tmLibdevErr_t**.

### The Standard Device Library API

The standard device library API is effectively the same as the standard TriMedia Software Architecture library API.

**Table 2** Basic TSA Standard Function API

| **Function** | **Description** |
|---|---|
| devGetCapabilities | Request the device capabilities, including the number of supported instances, the version number, and other device-specific information such as supported data formats, and so on. |
| devOpen | Requests an instance. That is, **devOpen** requests that the caller be added to the users of the device. |
| devInstanceSetup | Sets up the device (and board if necessary) for a certain instance. |
| devStart | Starts running the setup of the current instance. |
| devStop | Stops running the setup of the current instance. |
| devClose | Releases the instance. |

All functions return an error code of the type **tmLibdevErr_t**. **TMLIBDEV_OK** is defined as zero; it is returned when no error is encountered. As with any TSA compliant library, non-zero error codes are divided into module codes (high 16 bits), and specific errors. The module codes are defined in the include file tm1/tmLibdevErr.h. To minimize the execution time and code size overhead of the default library, asserts are used in the debug version of the library. The asserts primarily check alignment and size restrictions imposed by the hardware, and whether the passed instance is an owner of the device at that point. Philips recommends that you develop with the debug version of the library and then switch to the smaller and faster default library after the development phase.

More information about these functions is contained in the following pages.

Device library functions execute by default in the endianness in which the processor is executing.

Device library components do not:

- Claim any memory, though a device may use some static data.
- Depend on the presence or absence of a particular host.
- Depend on a specific board implementation.
- Take care of cache coherency.
- Specify an interrupt service routine (ISR).

## Board Support

The TriMedia Software Architecture includes two important libraries that achieve the design goals of the device libraries. These libraries are the registry and the component manager. Together, they allow system designers to build a board support package in such a way as to provide support for the standard (and non-standard) device libraries.

### Registry

The TriMedia registry (documented in Chapter 2, *TriMedia Registry Manager API*, of Book 5, *System Utilities*, Part A) provides a general way for a device library to find board-specific information. This information might be as basic as the presence or absence of a component. It might also be a set of function pointers that are used to implement the lower device library interface, that is the interface to the board. The registry is conceptually very simple. It is just a library that stores and retrieves information (in the form of 32-bit numbers), given "keys," in the form of text strings. The registry organizes these keys hierarchically, so you can look up something like **bsp/boardID** to find the value of the **boardID** variable.

### Component Manager

The registry provides a way in which libraries and applications can store and retrieve all sorts of information. The component manager provides a way for a system to install what may be board- or platform-dependent information into the registry. There are, of course, many ways that these variables could be added. You could call a function at the start of main(). You could initialize each component on its first use. But these approaches each have drawbacks. In order to meet the design goals of the device library, TriMedia uses a component manager that can initialize components before the user's program starts.

The technology of the component manager is related to that used in a linker. Components, as described here, are modules that can depend on other modules, and which are capable of exporting information on which other modules might depend. Using this information, the component manager can build a dependency tree and use it to initialize an arbitrary collection of components in a reasonable order.

As an example, consider the 'audio out' device library. This library needs a pointer to a function table from the registry. This registry entry must be made before audio out functions are called. Through the mechanism of the component manager, it is made before the call to the user's **main** routine. Since the audio out component is part of what has traditionally been called the BSP (Board Support Package), this initialization happens as part of the **boardInit** sequence. While more detail is provided in Chapter 19, *TMBoard API* of Book 5, *System Utilities,* Part C, a brief outline of the procedure is provided here.

## Board Bootup Sequence

When you build a program to run on one or more boards, you link in the appropriate board support packages. If your developers have different boards or platforms, it may be very helpful to build your program to run on several of them. When you build for a release, you will probably link in support only for the actual supported boards.

A board support package must export at least one output, using a macro like this,

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

as documented in Chapter 19, *TMBoard API* of Book 5, *System Utilities,* Part C. This exports the "activate" function, keyed to the board ID. Because bsp/boardID is an output with no matching input, the component manager will call it at the start of the initialization sequence. Each component has an activate function, and this function is what is called by the component manager. The job of the activate function is two-fold. First, activate will return True if the hardware supporting this component is found to exist. Second, the activate function will, on return, have performed any initialization required for the higher level libraries supported by this component. This probably means that the activate function put some information into the registry.

## Selecting Boards

Using this mechanism, a user can add or delete supported boards when he links his program. The tmconfig file used by **tmcc** includes variables which define a default list of boards. Users may want to empty this list, and instead, explicitly list the boards that they wish to support in their makefile. More documentation about this is contained in Chapter 3, *TriMedia Component Manager API*, of Book 5, *System Utilities*, Part A.

## Device Library Versions

The SDK includes APIs to support the devices that are available on the TM-1000, TM-1100, TM-1300, and TM-2700. The device library interface allows you to set up the devices in commonly used modes and to initialize board peripherals as necessary. The interface supports multiple instances for devices that permit multiple users.

Most device libraries are contained in the archive, libdev.a. Some of the basic supporting libraries are contained in libam.a, as they are needed by lower levels of the run-time system. Each of the device libraries is also provided as in a dynamically linked version (DLL).

### Debug Version

A precompiled version of the device library, with debug information included, contains **assert**s to allow additional error checking on alignment restrictions, and so on. You can use this version with the debugger, **tmdbg**, since it was compiled with the options **-g** and **-O2**. (The default library, libdev.a, is compiled with the **-O2** and **-DNDEBUG** options.) The sources are included in the toolset. **tmdbg** requires only the path to these sources.

You can link a program to the debug version of the library by specifying the option **-ldev_g** in the link line. This will ensure that libdev_g.a is linked in before the default libdev.a. For example, the command line:

```
tmcc vitest.c -o vitest.out -g -ldev_g
```

will create a version of the video in example program, vitest, linked with the debug library. The debugger can then be used to single-step through the test program and device libraries.

### Dynamic Linked Library Versions

All device library components have a corresponding dynamically linked library (DLL). A link option command file is automatically passed by the compiler driver **tmcc** to **tmld**. It enumerates all the DLLs, so the user need only specify the link type **app** or **dll**.

For example, the command line:

```
tmcc -btype app avio.c -o avio.app
```

will create a dynamically loadable version of the example program avio. The avio program uses Audio In/Out and Video In/Out in a pass-through mode and will reference, among others, the dynamic libraries libai.dll, libao.dll, libvi.dll, libvo.dll, and libiic.dll.

## Generic Function Prototypes

Each device implements what can be thought of as the device library “base class.” This is made up of the set of functions described in Table 2.

### devGetCapabilities

Each device has its own capability structure. A common part of this structure is defined as:

```
typedef struct {
   tmVersion_t   versionNum;
   Int           numSupportedInstances;
   Int           numCurrentInstances;
} devCapabilities_t, *pdevCapabilities_t;
```

The definition for **tmVersion_t** is located in tmlib/tmtypes.h.

Some peripherals store additional information in the capability structure. In particular, devices with coexisting devices on the board (such as the Audio In device and the ad1847 board component) store the name of the codec, as well as information about what data streaming formats they support. This will be explained further in the device-specific sections of this manual.

You can inspect the current status of any device via a call to the appropriate **devGetCapabilities** function:

```
tmLibdevErr_t devGetCapabilities( devCapabilities_t **pCap );
```

The function **devGetCapabilities** sets the pointer **pCap** to the device capability structure. This structure is read only for the user of the library. **devGetCapabilities** returns **TMLIBDEV_OK** upon success.

### devOpen

The function **devOpen** opens a device for initial use. It arbitrates and assigns an ownership. Its prototype is:

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

The instance returned by **devOpen** is read only for the user of the device library and must be passed to all subsequent device library calls. **devOpen** returns **TMLIBDEV_OK** upon success, or **TMLIBDEV_ERR_NO_MORE_INSTANCES** if no instances are available.

Some devices reset the device in the **devOpen** call or perform other initializations such reserving the interrupt vector. Sharable devices (IIC, ICP, and DMA) can handle multiple instances.

## devClose

The **devClose** function releases the claim on the device and frees the interrupt allocation. Its prototype is:

```
tmLibdevErr_t devStop( Int instance );
```

If the device is still running when **devClose** is called, it will be stopped.

The function **devClose** returns **TMLIBDEV_OK** upon success, or **TMLIBDEV_ERR_NOT_OWNER** if passed an invalid instance.

## devInstanceSetup

The **devInstanceSetup** function sets up the device as well as the board component needed for operating the device and performs necessary error checking on incoming parameters. It accepts an instance and a structure defining the necessary initializations:

```
tmLibdevErr_t devInstanceSetup( Int instance, devInstanceSetup *setup );
```

The function **devInstanceSetup** returns **TMLIBDEV_OK** upon success, or **TMLIBDEV_ERR_NOT_OWNER** if passed an invalid instance.

Once set up, the device can be started and stopped. Philips provides additional functions to change some parameters (such as buffer pointers) during device operation, so there is no need to call **devInstanceSetup** more than once.

## devStart

The **devStart** function starts running the device after **devInstanceSetup**. Its prototype is:

```
tmLibdevErr_t devStart( Int instance );
```

The function **devStart** returns **TMLIBDEV_OK** upon success, or **TMLIBDEV_ERR_NOT_OWNER** if passed an invalid instance.

## devStop

The **devStop** function stops the device after **devStart**. Its prototype is:

```
tmLibdevErr_t devStop( Int instance );
```

The function **devStop** returns **TMLIBDEV_OK** upon success, or **TMLIBDEV_ERR_NOT_OWNER** if passed an invalid instance.

## Generic Data Structures

### devCapabilities_t

All devices have a struct named **devCapabilities_t.** This struct will contain at least the following fields:

| | |
|---|---|
| `version` | The version of the installed device library module. |
| `numSupportedInstances` | The number of instances of the device which can be currently running. Some devices will allow only one device to run at a time whereas others will allow more. |
| `numCurrentInstances` | The number of instances of the device which are currently open. This number is incremented by the successful completion of the **devOpen** function and decremented by the successful completion of the **devClose** function. |

In addition to these fields, each device-specific struct will contain fields for device-specific information. For example, the **ai** struct (**aiCapabilities_t**) contains a field named **codecName**, among others. The **devCapabilities_t** struct may be accessed through the function **devGetCapabilities**.

### devInstanceSetup_t

This struct is common to all devices. It contains device-specific fields which allow the specific device to be initially set up (before the **devStart** function is called). It contains such fields as callback function addresses, interrupt enable flags, buffer sizes and locations, etc. It is passed as a parameter to the function **devInstanceSetup**.

Chapter 6

# pSOS+™ Real-Time Operating System

## Introduction

The pSOS™ operating system, from Integrated Systems, Inc. (ISI), is a modular, high-performance, real-time operating system (RTOS) designed specifically for embedded microprocessors. It provides a complete multitasking environment based on open system standards. pSOS employs a modular architecture, built around a real-time multitasking kernel and a collection of companion software components.

Although other operating systems can be ported to run on TriMedia, pSOS is the only operating system that Philips, under license agreement with ISI, has ported and standardized for TriMedia. pSOS+/TriMedia Version 1.1. is derived from pSOS+/PPC V2.0.6. For details concerning pSOS, not specific to TriMedia, refer to the documents *pSOS System Concepts* and *pSOS System Calls*, which are bundled on the CD.

This chapter describes the pSOS real-time operating system as it is used on TriMedia. It includes sections on compiling, and debugging pSOS applications, as well as pSOS+m™ (the multiprocessing version of the pSOS kernel), TriMedia support for multiprocessing on different platforms, and TriMedia networking components for pSOS. Note that, in this document, *pSOS* will refer to both the pSOS+ and the pSOS+m kernels. Also, in this document, when referring to directory path names, **$ (TCS)** refers to the TCS installation directory and **$ (PSOS_SYSTEM)** refers to **$ (TCS) /OS/pSOS/pSOSystem**. pSOS, pSOS+, and pSOS+m are registered trademarks of Integrated Systems, Inc.

Refer to the release notes for information about known bugs.

**Note**
TriMedia's pSOS is distributed and installed with the TriMedia (SDE) Software Development Environment. pSOS does not have to be installed separately.

## pSOS Application Structure

The pSOS kernel consists of various system calls that can be used by a pSOS application. The system calls provide functionality for task management, semaphores, message queues, dynamic memory allocation, time management, I/O functions, event macros, asynchronous signals (pSOS+m only), and fatal error handling. Refer to *pSOS System Calls* for detailed information about each system call.

Most of the pSOS kernel is provided as a library that can be linked into a user's application. The kernel is configurable by means of an include file. This include file (sys_conf.h) is compiled with a portion of the kernel, known as the pSOS board support package. By this mechanism, the pSOS kernel is tailored to the needs of each application.

Starting a pSOS application from scratch can be most efficiently achieved by copying and adapting one of the provided example directories, such as **$ (TCS) /examples/psos/psos_demo1**.

This directory reflects the structure of a minimal pSOS application, the parts of which are discussed in the following sections. In addition, many of the TriMedia application libraries make use of pSOS.

## root.c

This file contains the prescribed function named **root** for initializing and starting user execution, after initialization of pSOS has finished. It is responsible for creating all other pSOS objects that are required for the particular application, usually including message queues, semaphores, and other tasks.

The **root** function is executed by one of the two tasks that are created by pSOS itself during initialization. (The other one is the idle task). It has a high execution priority to minimize delay in application startup caused by other tasks being created.

Because the root task is a task like any other, the application can choose to let it participate with other created tasks in application execution after it completes its job as application initializer. However, many applications just let it suspend or terminate itself.

The **root** function is the usual place for initializing the pSOS device drivers. Because TriMedia devices are accessed using the TriMedia device library, usually the only device driver to initialize is that of the pSOS system timer, which is needed for the timed-event library and for task timeslicing to work. Hence, a minimal **root** function is the following:

```
void root(void){
   void *dummy;
   ULONG ioretval;

   de_init(DEV_TIMER, 0, &ioretval, &dummy);
   printf("Hello world\n");
   t_delete(0L);
}
```

root.c can be adapted to create a new pSOS application.

## drv_conf.c

drv_conf.c is the means by which pSOS drivers are customized. Because many TriMedia and TSSA programs do not use pSOS drivers, a generic default copy of drv_conf.c is normally linked into the pSOS board support package.

This file contains the second function that applications should provide. It has the prescribed name **SetUpDrivers**. As opposed to **root**, this function is called very early during pSOS initialization, in a stage at which the total memory assigned to pSOS is being divided between pSOS components (like pNA) and pSOS device drivers. No tasks are executed yet, and none of the regular pSOS functions will work yet.

**SetUpDrivers** is the place where the application should install the pSOS drivers that it wants to use, and to optionally reserve memory needed by these drivers. When no specific pSOS drivers are needed, this file can be left untouched. It contains conditionalized installations of many standard device drivers provided by ISI. In some cases, it may even be an empty function.

**SetUpDrivers** installs information for drivers in the pSOS I/O table. The function **InstallDriver** is called to add each driver. **InstallDriver** takes the following arguments:

| Argument | Meaning |
|---|---|
| USHORT major_number | Device major number |
| void (*dev_init)() | Device init procedure |
| void (*dev_open)() | Device open procedure |
| void (*dev_close)() | Device close procedure |
| void (*dev_read)() | Device read procedure |
| void (*dev_write)() | Device write procedure |
| void (*dev_ioctl)() | Device control procedure |
| ULONG rsvd1 | Reserved |
| ULONG rsvd2 | Reserved |

If you are adding a driver that must be initialized before pSOS drivers are initialized, or before a driver's init function is called, you can create a local copy of drv_conf.c and modify it to call a setup function for the driver. For example, refer to **CnslSetup** in **SetUpDrivers** for a serial device driver. If the initialization function must allocate memory, pass **FreeMem** to it as an argument, the same as the other initialization functions in **SetUpDrivers**, which will return what is left of the memory for the next setup routine.

## pSOS Board Support Package

The pSOS board support package (pSOS BSP) traditionally contains the implementations of the hardware-specific software available to the application. This consists of the device drivers (Note: *drv_conf.c* contained only the enabling of these drivers), pSOS boot code, pSOS configuration code, and hardware-access libraries such as those needed for installing interrupt handlers and system timers. It is a repository of hardware and configuration functions that is shared by many applications, and, therefore, it is part of the pSOS installation under the pSOSystem directory, rather than part of the application code.

**IMPORTANT**
The pSOS BSP is different from the TriMedia device library BSP.

In TriMedia pSOS, all relevant hardware-access functions are already available in the TriMedia device library, and no pSOS device drivers have been delivered. Also, pSOS starts up as a normal **main** function, using the TCS boot code like any other TriMedia application. For these reasons, the function of the pSOS BSP has been reduced to pSOS kernel configuration only.

Kernel configuration is performed by a number of functions that pass the pSOS configuration parameters defined in sys_conf.h via tables to the pSOS kernel, and the pSOS **main** function, which initializes and starts this kernel.

Because it compiles the application-specific configuration parameters into the pSOS kernel, the configuration sources should be recompiled for each application; hence, it cannot be devised as a precompiled library.

There usually is no need for users to interfere with the pSOS BSP/pSOS configuration code.

## pSOS Kernel

The pSOS Kernel comes in numerous configurations, depending on endianness, the use of dynamic linking, and the use of multiprocessors. The objects that contain the various versions of the pSOS kernel are *psos_tm_eb.o*, *psos_tm_el.o*, *psos_tm_eb.dll*, *psos_tm_el.dll*, *psosm_tm_eb.o*, *psosm_tm_el.o*, *psosm_tm_eb.dll*, and *psosm_tm_el.dll*. They can also be found in the pSOSystem directory.

## sys_conf.h

sys_conf.h is a configuration file containing macros by which a pSOS application can be configured. Macros are available for enabling/disabling pSOS components (when supported), for enabling/disabling the multiprocessor extension (pSOS+m), for resizing the various preallocated resource sets (task descriptors, objects, message blocks), and for installing user definable callback functions into, for example, the scheduler. Refer to Figure 8 below.

Since basic programs do not often change sys_conf.h, a default version is often used. This file is used to build the pSOS board support package. If you want to customize sys_conf.h, create a local copy and customize it as necessary.

Set the macros **KC_SYSSTK, KC_ROOTSSTK**, and **KC_ROOTUSTK** to at least 8K to prevent stack overflow. Usually, 8K is high enough, but, in specific cases, these need to be set higher.

**KC_NTASK**, **KC_NQUEUE**, **KC_NSEMA4**, **KC_NMSGBUF**, and **KC_NTIMER** configure the number of “local objects” that exist in the system. If these are too low, the creation of a task or a queue may fail. Hence, it is critical that the error codes are checked when creating pSOS objects.

Definitions are given for a number of “callout” functions. Of particular interest here are the task switch callout and the fatal error handler. The task switch callout can be used to trace the execution of a pSOS application. This is an example of a task switch callout:

```
void task_switch (unsigned long entering_tid, void* entering_tcb,
                  unsigned long  leaving_tid,  void* leaving_tcb){
```

```
   DP((" Leaving task %x. ", leaving_tid));
   DP((" Entering: task %x.  \n", entering_tid));
}
```

Also useful is the fatal error callout:

**extern void fatal_handler( unsigned long err, unsigned long flag );**

Stack overflow detection has been added to TriMedia's port of pSOS. The macro, **SC_STACK_OVF_CHECK**, in sys_conf.h, defines the number of bytes at the end of the stack (of each task) that will be filled with a known pattern, and that will be checked for being overwritten at each system call. The default number of bytes to be checked is 8. When an overwrite is detected, pSOS execution will be aborted with fatal error **FAT_STKOVF (0xF30)**. Note that **SC_STACK_OVF_CHECK** will be truncated to an integral number of words.

**SC_RAM_SIZE** is no longer used by the TriMedia port of pSOS. At the end of initiation, pSOS claims the maximum available memory remaining on the board for dynamic allocation in "Region 0." No change is required by the user for this macro.

A macro, **TCS_MALLOC_USE**, has been added to sys_conf.h. When it is enabled, the TCS memory manager will be used for malloc/free. When disabled, malloc/free will be mapped to **rn_getseg**/**rn_free** from Region 0, as is standard in pSOS. The pSOS region manager might be more predictable in its real-time behavior, but this is at the cost of larger unit sizes. The Region 0 unit size can be adjusted by changing **KC_RN0USIZE**. Also, the pSOS region manager cannot hold more than 32K units, which is 8MB with the current **KC_RN0USIZE**, but proportionally less when the unit size is decreased.

When **TCS_MALLOC_USE** is enabled, you must define **TCS_REGION0_SIZE** such that Region 0 does not occupy all free memory at the end of pSOS initialization. When not defined, all free memory (limited to 32K units) is given to Region 0. Otherwise, Region 0 is created with the specified size, also limited to 32K units, and all other memory is available via the TCS memory manager. Use this macro in combination with **TCS_MALLOC_USE** and **KC_RN0USIZE**, when **KC_RN0USIZE** result in a Region 0 does not contain all available SDRAM.

In order to link new pSOS components ported by TriMedia, such as pSOS+m and pNA, to your pSOS applications, move the macros, **SC_PSOS**, **SC_PSOSM**, and **SC_PNA** from sys_conf.h to the pSOS application makefile.  **SC_PSOS** and **SC_PSOSM** will be set according to the value of the PSOS macro. Similarly, **SC_PNA** will be set according to the value of the PNA macro (see *pSOS Example Makefile* below).

```
/*---------------------------------------------------------------------*/
/* pSOS+ configuration parameters                                      */
/*---------------------------------------------------------------------*/
   #define KC_RN0USIZE    0x100    /* region 0 unit size               */
   #define KC_NTASK       12       /* max number of tasks              */
   #define KC_NQUEUE      10       /* max number of message queues     */
   #define KC_NSEMA4      30       /* max number of semaphores         */
   #define KC_NMSGBUF     100      /* max number of message buffers    */
   #define KC_NTIMER      10       /* max number of timers             */
   #define KC_NLOCOBJ     50       /* max number of local objects      */
   #define KC_TICKS2SEC   100      /* clock tick interrupt frequency   */
   #define KC_TICKS2SLICE 10       /* time slice quantum, in ticks     */
   #define KC_SYSSTK      0x1000   /* pSOS+ system stack size (bytes)  */
   #define KC_ROOTSSTK    0x1000   /* ROOT supervisor stack size       */
   #define KC_ROOTUSTK    0x1000   /* ROOT user stack size             */
   #define KC_ROOTMODE    0x2000   /* ROOT initial mode                */

 /*-------------------------------------------------------------------*/
 /* The following are examples for modifying the following defines   */
 /*                                                                   */
 /* Using a pSOSystem routine as a fatal error handler                */
 /* #define KC_FATAL    ((void (*)()) SysInitFail)                    */
 /*                                                                   */
 /* Using a user written routine as a fatal error handler             */
 /* extern void MyHandler (void);                                     */
 /* #define KC_FATAL    ((void (*)()) MyHandler)                      */
 /*                                                                   */
 /*-------------------------------------------------------------------*/

   #define KC_STARTCO     0        /* callout at task activation       */
   #define KC_DELETECO    0        /* callout at task deletion         */
   #define KC_SWITCHCO    0        /* callout at task switch           */
   #define KC_FATAL       0        /* fatal error handler address      */
   #define KC_ROOTPRI     230      /* ROOT task priority               */

/* NB: The following macros have been moved from sys_conf.h to the pSOS demo
   makefile which is discussed in the next section. Errors might result when
   using this demo makefile in combination with a sys_conf.h which still
   defines these macros, or when using a sysconf.h intended for the demo
   makefile in combination with another makefile:*/

   #define SC_PSOS YES  /* pSOS+ real-time kernel                   */
   #define SC_PSOSM NO /* pSOS+ real-time multiprocessing kernel  */
   #define SC_PNA NO   /* pNA+ TCP/IP networking manager           */
```

**Figure 8** The sys_conf.h configuration file.

## Other pSOS Components

ISI provides a relatively wide selection of pSOS components. A small number of these have been ported to TriMedia.

- pNA is the pSOS network stack. This has been ported, and it is included with the release.
- pROBE is the pSOS debugger. Most of the functionality of pROBE has been incorporated into the TriMedia debugger.

## pSOS Example Makefile

pSOS applications can be compiled in a number of configurations, enabling or disabling options such as multiprocessing, dynamic linking, and pSOS network components, and compiling for different hosts and endianness.

Different sets of pSOS binaries must be linked, depending on the options enabled, and different definitions must be provided. The makefiles hide users from the details, by providing macros and allowing them to quickly switch between different application configurations by redefining certain macros.

The complete pSOS makefile is large enough to prove daunting to the casual observer. For the users who do not need the different configuration options, a simplified version of the makefile is provided. Both makefiles can be found in **$ (TCS) /examples/psos/ *demo*** (*Makefile* and *Makefile.simple*). The simple makefile removed the options to enable or disable multiprocessing, dynamic linking, and pSOS network components (dynlink_demo uses dynamic linking).

However, a multiprocessing version of this makefile can be found in **$ (TCS) /examples/ psos/psos_mp_demo1**. When using *Makefile.simple*, a make clean is necessary after redefining any of the macros. The simple makefile can be ported to be used with Microsoft's NMAKE, while the original makefile cannot because of the use of nested macros. See the comments at the top of Makefile.simple for instructions on how to port it to NMAKE.

The simple *Makefile* is used in the examples that use pSOS in **$ (TCS) /examples** as *Makefile*. The examples that use pSOS in peripherals include: *patest*, *vrend*, and *vtrans*. The miscellaneous examples include: *dynamic_loader_shell, psos_files,* and *tipc*; in multiprocessor, **data_streamer**.

After setting the macros to the desired configuration in the makefile you want to use, run them in a Korn Shell from the MKS Toolkit by typing

```
make
```

to use *Makefile*, or by typing

```
make -f Makefile.simple
```

to use *Makefile.simple*.

The simple makefile is small enough to be self-explanatory.

## The Complete pSOS Makefile

For simple pSOS applications directly derived from

$ (TCS) /examples/psos/*demo*

(and for these demos themselves), the makefile can be made to work by simply defining the macro TCS to the actual TCS installation. This defines the compiler version to be used, as well as the location of the pSOSystem installation.

A default rule is provided for compiling any C file that is located in the current directory in which the makefile is invoked. Relying on this rule, new application source files can be provided for in the makefile by simply extending the **OBJECTS** macro with the corresponding object file names. For instance, a new file called *new.c* in the current directory can be provided for in the makefile by simply adding **$ (OBJDIR) /new.o**.

Compilation directories named OBJDIR_‹host›_‹endian› and PSOS_CONFIG_‹host›_‹endian› will hold the generated object files for the <host>/<endian> combination chosen. They can be removed using a “make clean.”

The following describes the different parts of the demo makefile.

```
##############################################################################
    # Location of compiler
##############################################################################

    TCS         = /t/qasoft/build/SunOS
    PSOS_SYSTEM = $(PSOS_SYSTEM)
```

These macros define the location of the TCS compiler installation, including pSOS. Since **PSOS_SYSTEM** is defined in terms of TCS, usually only the **TCS** macro must be adapted.

**TCS** probably is the only macro that you must change when you encounter this makefile for the first time.

```
##############################################################################
    # Compilation- and link flags
##############################################################################

    CINCS    = -I. \
               -I$(PSOS_SYSTEM)/include
    CFLAGS   =
               # code compaction optimizations by tmld:
    LDFLAGS = -bremoveunusedcode -bcompact -bfoldcode
```

These macros can be changed to pass new include paths or compilation macros to source files that are compiled using the default compilation rule (in the following paragraph), or to use different linker options when linking the application.

The **LDFLAGS** as shown pass the full set of code compaction options to the linker. Apart from reducing the application code size by sharing identical dtrees, such as common epilogues of "C" functions, and by reordering dtrees to minimize instruction padding, the -**bremoveunusedcode** macro has the effect of removing unused parts of pSOS. For instance, an application that only makes use of multitasking and message passing via queues will not get a copy of, for example, the pSOS semaphore or timer libraries.

```
##############################################################################
    # Desired name of application, plus objects to link
##############################################################################

    APPLICATION = data_streamer.out

    OBJECTS= \
             $(OBJDIR)/root.o \
             $(OBJDIR)/drv_conf.o

    target: $(APPLICATION)
```

This defines the name of the application, that is, the result produced by executing this makefile, and the object files from which it should be linked. This list of object files should not include the pSOS kernel object as described in the earlier section *pSOS Application Structure*, nor the files compiled from the pSOS configuration files.

As described earlier in this section, new C files to be compiled with the application can be incorporated in the makefile by adding a corresponding entry in the **OBJECTS** list. When these C files are placed in the current directory, then the default make rule described under discussion of the “Application Building” portion of the file later in this section will automatically take care of compiling them into object files.

This is already illustrated by the treatment of the files *root.c*. These files occur in the same directory as the makefile itself, and they are compiled into object files by the default make rule.

Note that all objects are to be placed in a subdirectory that is referred to by the macro **OBJDIR**. This prevents the source directory from being trashed with intermediate files.

```
####################################################################
    # Selected application configuration
####################################################################

    HOST  = tmsim
    #HOST = MacOS
    #HOST = Win95
    #HOST = WinNT
    #HOST = nohost

    DYNAMIC  = nodynamic
    #DYNAMIC = dynamic

    ENDIAN    = el
    #ENDIAN   = eb

    #PSOS  = psosm
    #PNA   = pna
    #PPP   = ppp
```

This list of macros allow users to specify the application configuration: TriMedia execution host, whether the use of the dynamic linker should be allowed, required TriMedia endianness, and the required list of pSOS extensions.

Note the following remarks:

- **ENDIAN=eb** does not work in combination with **HOST=Win95**. The other HOST options are compatible with either endianness.
- Selection of **DYNAMIC=dynamic** causes the resulting **APPLICATION** to become a dynboot application, containing pSOS as an embedded dynamic library. A dynboot contains the dynamic loader, and hence is able to load task code at run time. When such loaded task code has references to pSOS, then linking pSOS as an embedded dynamic library will allow such task code to detect it during loading. An example of this is discussed in detail in the Chapter 13, *Dynamic Linking API*, of Book 5, *System Utilities*. Unused pSOS code removal is far less effective in the case of the use of pSOS+m, or in the case of dynamic loading. The reason for this is that external calls to pSOS can be made in either of these combinations, from arbitrary dynamically loaded code or from other pSOS nodes, so that it is impossible to predict whether a particular pSOS function will be used.
- Creating an embedded, standalone application is possible simply by using **HOST=nohost**. ANSI file I/O is not possible in such a configuration, that is, calls to **printf** will remain possible, but they will return a failure status unless a new I/O driver

has been installed, and the application will be stripped from the generic host communication library.

- Despite the name, applications created with **HOST=nohost** will run on any of the supported TCS platforms except that ANSI I/O will not work. This means that standalone applications that have their own means of doing I/O, possibly through the TriMedia devices, and can be tested on a MacOS or on a PC-hosted TriMedia board or under **tmsim**.
- pSOS+m based (multiprocessor) applications should be started using **tmmprun**, which downloads executables to the requested TriMedia nodes, and which assigns node identifications to each of the nodes. Because **tmmprun** assigns these numbers, they need not be defined at application compile time, so that the same executable can be used for more than one node or even for all of the nodes.

```
#######################################################################
    # Include invariant part of this makefile
#######################################################################

    include $(PSOS_SYSTEM)/include/Makefile.inc
```

```
#######################################################################
    # Application building
#######################################################################

    $(OBJDIR)/%o: %c
            @ echo "Compiling $(*)c"
            $(CC) $(CFLAGS) $(CINCS) -c $(*)c -o $@
            $(APPLICATION) : $(CHECK) .$(PSOS_CONFIG) $(OBJECTS) Makefile
            @ echo "Linking $(APPLICATION)"
            $(CC) \
            $(OBJECTS) $(PSOS_LINK) $(PSOS_CONFIG)/bsp.a \
            $(LDFLAGS) $(CFLAGS) -o $(APPLICATION)
```

This include file isolates several technical definitions (which strongly depend on the chosen configuration parameters) of the following:

- **PSOS_CONFIG** defines the compilation directory for the pSOS BSP (see the earlier section *pSOS Application Structure*).
- **OBJDIR** defines the compilation directory for all sources compiled by the default make rule described below.
- The **PSOS_LINK** macro must be included in the link command line for linking **APPLICATION** (see rule below). It contains the proper link macros and it contains the proper versions of all enabled pSOS components.
- The **PSOS_OBJECT** macro is only needed when creating a code segment that is to be dynamically loaded by **APPLICATION** (note that **DYNAMIC= dynamic** in such cases). The macro contains the names of all pSOS dynamic libraries, so that the references to pSOS can be resolved by the linker. See the dynamic loader shell example described later in this document.
- For make technical reasons, the directories **PSOS_CONFIG** and **OBJDIR** have timestamp files **PSOS_CONFIG** and **OBJDIR** associated.

- **PSOS_CONFIG** and **OBJDIR** are dependent on the selected **HOST/ENDIAN** values. This allows switching between different values of **HOST** or **ENDIAN** without having to do a clean build.

The previous make rules define how to build the application, and how to create the object file mentioned in the **OBJECTS** macro from a corresponding C file in the current directory.

```
####################################################################################
    # Cleanup
####################################################################################

    clean :; rm -rf *.o *.a *% $(APPLICATION)   \
                            .PSOS_CONFIG* PSOS_CONFIG* \
                            .OBJDIR*      OBJDIR*
```

## pSOS+m on TriMedia

### Introduction to pSOS+m

pSOS+m, the multiprocessing version of the pSOS kernel, extends most of the pSOS+ system calls to operate seamlessly across multiple processors and also adds some functionality relevant only to multiprocessor systems. As pSOS+m is designed for functionally divided multiprocessing systems, it is especially suitable for real-time applications using TriMedia. For more details behind the concepts involved in pSOS+m, refer to Chapter 3, *pSOS+m Multiprocessing Kernel*, in the *pSOS System Concepts* document.

### Implementation of pSOS+m

TriMedia's pSOS+m is implemented using shared memory across the PCI bus. As the pSOS+m kernel itself is designed to be independent of the physical medium connecting the various processors, it relies on the standard API in the Kernel Interface (KI) to provide the actual shared memory connection.

### Necessary Changes to Use pSOS+m

To use pSOS+m, you must make two changes for the compiling process. First, change the **PSOS** macro in the application makefile to **psosm**.

Second, modify the definitions of **SD_SM_NODE** and **SD_KISM** in the sys_conf.h file, by replacing them with the following:

```
extern int _node_number;
#define SD_SM_NODE (_node_number+1)

extern int _number_of_nodes;
#define SD_KISM  _number_of_nodes
```

### Node Numbering

Notice that **SD_SM_NODE** is **_node_number** plus 1. The reason for this is that the node numbering for pSOS+m starts at 1, while the node numbering for TriMedia start at 0. From now on, this document will use the TriMedia system for node numbering. The values of **_node_number** and **_number_of_nodes** will be filled in by the downloader and, therefore, do not have to be hard-coded in *sys_conf.h*.

## TriMedia Support for Multiprocessors

The following example assumes that you have at least three TM1s in the machine and the names of your applications are a.out, b.out, and c.out. You can also load the same application on more than one TM1.

Note that you must load your applications in the order you intended for node numbering. For example, *a.out* will be run the master node (node #0), and *b.out* and *c.out* will be run on node #1 and node #2, respectively.

### Windows

TriMedia provides support for multiprocessing on Windows with the tool **tmmprun**. To run several applications on multiple TM1's, type the following line:

```
> tmmprun -exec a.out -exec b.out -exec c.out
```

### Shared Memory Support in tmcc

Shared memory can be used in applications without using pSOS+m. The only drawback in using shared memory instead of using pSOS+m system calls to communicate across processors is that the user must provide synchronization for data access.

If you want to use shared memory, TriMedia provides options in **tmcc** to compile data into shared memory. First, declare the uninitialized data you want in shared memory into one file. You must use the actual structure or type declarations, as opposed to pointers to the structures or types. Remember that any data you include in another file in the shared memory file will also be put into shared memory, so do not include a file that declares data in the shared memory file unless you want its data to be put into shared memory, also. This example, calls the shared memory file *usershared.c*, contains only the following variable declaration:

```
unsigned long shared_mem[1000];
```

The following makefile rule will create a usershared.o, which should be linked to each application that will be run on different nodes.

**$ (OBJDIR) /usershared.o** should be added to the **OBJECTS** macro in the makefile as follows:

```
$(OBJDIR)/usershared.o: usershared.c
        $(CC) -c $(CFLAGS) $(CINCS) -o $(OBJDIR)/usershared.o  \
usershared.c
        $(LD) $(OBJDIR)/usershared.o -o $(OBJDIR)/usershared.o  \
-map_commons -sectionrename bss=shared  \
-sectionproperty  bss=shared -sectionproperty  \
bss=uncached
```

The following table explains the results:

| **Statement** | **Result** |
|---|---|
| bss | Represents the uninitialized data section. |
| map_commons | Maps common symbols to bss section. |
| sectionrename bss=shared | Renames the section to shared. |
| sectionproperty bss=shared | Sets shared property for the section. |
| sectionproperty bss=uncached | Sets uncached property for the section. |

For more information on the TriMedia linker, refer to Chapter 11, *Linking TriMedia Object Modules,* of Book 4, *Software Tools*, Part B.

## pSOS Networking Components

The PPP-TM Network Interface is an implementation of the Point-to-Point Protocol (PPP), as defined in RFC1331 and RFC1332. This pSOS networking component supports all Link Control Protocol (LCP) options except the Quality-Protocol. It also supports all IPCP options and provides configurability to work with deprecated options.

### What it Contains

The PPP-TM is implemented as a Network Interface (NI) to the pNA+ component to allow TCP/IP operations over serial lines. It can be extended to support other network layers, provided they observe the same NI used by pNA+. PPP-TM supports only asynchronous links. The underlying serial hardware must be full-duplex.

### PPP-TM Operations

The PPP protocol consists of the following four components:

- A method of encapsulating datagrams over serial links
- An LCP for establishing, configuring, and testing the data-link connection

- A suite of authentication protocols that contains Challenge Handshake Authentication Protocol (CHAP) and Password Authentication Protocol (PAP), that are used to authenticate the peer
- A family of Network Control Protocols (NCPs) for establishing and configuring different network layer protocols (for example, IP)

PPP TM supports one NCP, which is IPCP for the TCP/IP network layer.

## Configurations

You must configure several site-dependent parameters. These are defined in the *ppp_conf.h* files under the **$(PSOS_REL)/include** directory.

### PPP Operation Parameters

The following parameters are local-defined in the file *ppp_conf.h* and are not negotiated with the peer. They are site-dependent and should be tuned to get the best result according to the application environment. The values in parentheses are the defaults.

- **NPPPBUF (32)**—Number of PPP buffers. PPP maintains a pool of buffers for two purposes: to send PPP negotiation packets and to receive data. Once the link is established, all buffers in the pool can be used for receiving. Buffer size: **MAX(MYMRU, 1500)**.
- **DEBUG—(YES)**—This determines if diagnostic information should be dumped to the console.
- **DEFTIMEOUT (5 seconds)**—If PPP does not receive an ACK to either **CONFREQ** or **TERMREQ** during this time period, it retransmits the request.
- **DEFMAXCONFTRANSMITS(10)**— How many times PPP retransmits the **CONFREQ**s.
- **DEFMAXTERMTRANSMITS(10)**—How many times PPP retransmits the **TERMREQ**s.
- **DEFMAXNAKLOOPS (10)**—Number of retries upon receipt of **CONFNAK**.

### Configuration Table

Each PPP link requires a set of configurable parameters defined via the configuration table. The configuration table has default values and can be updated by the application. The table is a C structure containing the following entries:

```
struct ppp_cfg {
   unsigned long channel;      /* serial Channel number for PPP   */
   ChannelCfg ccfg;            /* serial channel configuration    */
   long pppmode;               /* Mode for the PPP channel        */
   long dialmode;              /* Dial mode for the channel       */
   char *setupscript;          /* setup script for dialup         */
   char *dialscript;           /* dial script for dialup          */
   char *hangupscript;         /* hangup script for dialup        */
   char *user;                 /* User                            */
   char *passwd;               /* Password                        */
```

```
    char *tel_number;           /* telephone number for dialup     */
    unsigned long dialtimeout;  /* demand dial timeout in minutes  */
    unsigned long mru;          /* MRU of the channel              */
    unsigned long asyncmap;     /* Async control char Map          */
    unsigned long lcp_options;  /* Various LCP options             */
    unsigned long auth_options; /* Various Auth Options            */
    unsigned long local_ip;     /* Local IP address of the channel */
    unsigned long peer_ip;      /* Peer IP address                 */
    unsigned long ipcp_options; /* Various IPCP options            */
  };
```

The following are parameters in this structure:

- **channel**—Serial Channel for PPP link.
- **cfg**—Serial Channel configuration
- **pppmode**—The mode of operation for the PPP link. If **pppmode** is set to **PPPMODE_ACTIVE**, then the link is in active mode, (that is, it initiates the PPP connection). If the mode is set to **PPPMODE_PASSIVE**, then the link is set to **PASSIVE** mode, (that is, it waits for connection initiation).
- **dialmode**—Sets the dial mode of the link. If the dialmode is set to **DIRECT**, then the link is a direct connection. If the dialmode is set to **DIALUP**, then the link is connected via a modem and the call setup procedure is initiated by using the modem scripts. If the dialmode is **DEMANDDIAL**, then the link is connected via a modem and it requires to be dialed on demand. A call setup procedure is initiated via the modem scripts and additionally the link is monitored for activity to bring it down upon a timeout.
- **setupscript**—Contains the script to initialize the modem when the **dialmode** is set to **DIALUP** or **DEMANDDIAL**.
- **dialscript**—Contains script to dial out via the modem. The **dialmode** should be set to **DIALUP** or **DEMANDDIAL** for this to be used.
- **hangupscript**—Contains script to hang up the telephone connection. This is used when the **dialmode** is set to **DIALUP** or **DEMANDDIAL**.
- **user**—Contains the user name used in the script substitution.
- **passwd**—Contains the password used in the script substitution.
- **tel_number**—Contains a telephone number used in the script substitution.
- **dialtimeout**—Timeout value used in demand dial. If the link is inactive for **dialtimeout** seconds, then the PPP connection is brought down.
- **mru**—**MRU** used for PPP link.
- **asyncmap**—A 32-bit value indicating ASCII values from 0 - 31. Each 32-bit position corresponds to one ASCII value. If a bit is set, its corresponding character must be ESCAPED. For example, if your serial hardware must use XON/XOFF, you might want to use an **MYASYNCMAP** with the 17th and 19th bit set.

- **lcp_options**: Contains various LCP options that are negotiated for PPP connection. The options are set by setting various bits. The following bits are used for the options:

```
#define NEGMRU          0x1     /* Negotiate MRU                */
#define NEGASYNCMAP     0x2     /* Negotiate async map          */
#define NEGMAGIC        0x4     /* Negotiate magic number       */
#define NEGPROTOCOMP    0x8     /*Negotiate protocol compression */
#define NEGACCOMP       0x10    /* Negotiate addr & control comp */
```

The following is true:

- **NEGMRU**—Is used to tell the peer the maximum size of a packet it can receive. The peer uses this information to calculate its maximum transmission unit (MTU). The buffer size of the local node is the larger of this value and 1500, since all PPPs are expected to receive packets up to 1500 bytes, if the negotiation fails.
- **NEGASYNCMAP**—negotiates the Async map with the peer.
- **NEGMAGIC**—negotiates the magic number. This is necessary to avoid loopback links.
- **NEGPROTOCOMP**—negotiates the Protocol-Field-Compression.
- **NEGACCOMP**—negotiates Address-and-Control-Field-Compression.
- **auth_options**—The desired **PAP**/**CHAP** authorization options.

```
#define REQUPAP   0x1    /* Negotiate PAP                  */
#define REQCHAP   0x2    /* Negotiate CHAP                 */
#define NOUPAP    0x4    /* Dont allow PAP authentication  */
#define NOCHAP    0x8    /* Dont allow CHAP authentication */
```

- **local_ip**—The desired local ip address of the link.
- **peer_ip**—The desired peer ip address of the link.
- **ipcp_options**—Sets up various IPCP options. The options are set using various bits. The following values are defined for IPCP options:

```
#define NEGADDR     0x1     /* Negotiate IPCP addr compression */
#define NEGIPCOMP   0x2     /* Negotiate IPCP compression      */
```

## NI Configuration Table

The PPP-TM NI is configured through the **add_ni**() call. This is a sample NI configuration table for PPP:

```
static struct ni_init ni_ppp[] = {
   (int (*)())NiPPP,                      /* ptr to interface code      */
   htonl(PPP_LOCAL_IP),                   /* IP address                 */
   DEFMRU,                                /* maximum transmission unit  */
   4,                                     /* length of hardware address */
   IFF_NOARP|IFF_POINTTOPOINT|IFF_RAWMEM, /* flags                      */
   0,                                     /* subnet mask                */
   htonl(PPP_PEER_IP),                    /* peer IP address            */
   0
};
```

To work with the zero-copy pNA+ feature, the **RAWMEM** bit must be set.

The following code segment illustrates the actual use of the PPP NI with **add_ni** call:

```
#include "ppp_conf.h"
extern long NiPPP();
extern unsigned long PPPNiNum;
#include <configs.h>
extern NODE_CT NodeCfg;
#define SEC2TICKS(sec) (NodeCfg.psosct->kc_ticks2sec * sec)

client(){
   struct ppp_ioctl pi;
   struct ifreq ifr;
   struct sockaddr_in *sin;
   /* ... other local variables */

/* Add PPP Ni.*/
   if( add_ni(ni_ppp) ) error("add_ni() error");

/* Clear out address structures.*/
   memset( (char*)&myaddr_in  , 0, sizeof(struct sockaddr_in) );
   memset( (char*)&peeraddr_in, 0, sizeof(struct sockaddr_in) );

/* Create the socket.*/
   s = socket(AF_INET, SOCK_STREAM, 0);
   if( s == -1 ) error("SOCKET creation error");

/* Need to wait till the PPP driver comes up.*/
   do{
      pi.pi_ifno = PPPNiNum;
      if( ioctl(s, SIOCGPPPSTATUS, (char *)&pi) < 0 )
            error("ioctl(SIOCGPPPSTATUS) error");
      if( pi.pi_status == PSDOWN ) error("PPP down!");
      tm_wkafter(SEC2TICKS(5));
   }while (pi.pi_status != PSUP);

/* Get the Peer IP address result from the negotiation.*/
   ifr.ifr_ifno = PPPNiNum;
   if( ioctl(s, SIOCGIFDSTADDR, (char *) &ifr) < 0 )
      error("ioctl (SIOCGIFDSTADDR)");
   else
      sin = (struct sockaddr_in *) (&ifr.ifr_dstaddr);

   peeraddr_in.sin_family = AF_INET;
   peeraddr_in.sin_addr.s_addr = sin->sin_addr.s_addr;
   peeraddr_in.sin_port = SERVER_PORT;

/* Try to connect to the remote server at the address in peeraddr_in. */
   rc = connect(s, &peeraddr_in, sizeof(peeraddr_in));
   if( rc == -1 ){
      close(0);
      error("CONNECT error");
   }

   /* ... more code */
}
```

## Error Handling

PPP-TM exports a global variable named **ppperrno** to indicate the error status. Currently, **ppperrno** reports two types of errors:

- **EMIB**—This error results when an **ifMIB** operation is attempted to the PPP driver when the **ifAdminstatus** is down and the operation is not to change the **ifAdminstatus**.
- **ETIMEOUT**—This error occurs when the maximum number of PPP config-request retransmissions is exceeded. When this happens, the **ioctl**(**SIOCGPPSTATUS**) should return a **PSDOWN** status.

## Building Applications with the PPP-TM

PPP makes use of a Device Independent Serial Interface (DISI) to communicate with the physical layer which could be a universal asynchronous transmitter/receiver (UART) or a modem. The physical layer, driver should provide the following functions:

| | |
|---|---|
| SerialSlipInit | Installs the transmit and receive interrupt handlers for a given channel. |
| SerialIntRead | Reads a character from a channel. |
| SerialIntWrite | Writes a character to a channel. |
| SerialIntTxion | Enables the transmit interrupt for a channel. |
| SerialIntTxioff | Disables the transmit interrupt for a channel. |
| SerialIntRxioff | Disables the Receive interrupt for a channel. |

The prototypes are defined in *~/PSOS_REL/include/bspfuncs.h*. The PPP-TM can support four serial channels simultaneously. It requires the transmit and receive interrupt handlers to be installed by the driver. These handlers should call the two ISRs viz., Stisr and Srisr, as demonstrated in *~/PSOS_REL/examples/ppp/driver.c file*. If the interrupts are multiplexed, a channel identifier must be passed to the ISRs. Otherwise, separate interrupt handlers should be installed using the **SerialSlipInit** call. You should keep the physical layer driver out of the pSOS I/O system to reduce run-time lengths for all driver functions.

In a local copy of the file drv_conf.c, add the following line to the end of the **SetUpNI** routine, where the comment says “Add additional network drivers here”:

```
#include "drv_conf.ppp"
```

In the makefile, set the following switches as

```
PNA=pna
PPP=ppp
```

Now the application defined in the makefile can be built by running it for the target and endianness selected.

## Debugging pSOS Applications on TriMedia

Most of the functionality of the pSOS pROBE debugger is available in the TriMedia debugger. For information on debugging pSOS Applications on TriMedia, see Book 4, *Software Tools,* Part C.

In most TriMedia code (at least, in code that complies with the TriMedia Software Architecture), pSOS function calls are not accessed directly. Instead, they are accessed through a wrapper layer known as tmospSOS. These wrapper functions make it easy for the underlying operating system to be replaced or emulated, if such a need should arise. The tmospSOS wrappers are documented in Chapter 9, *The Operating System Wrapper (tmos.h),* of Book 5, *System Utilities*, Part A. You will find them a straightforward mapping of a subset of pSOS.