*Book 2—Cookbook*

Part B:

# Programming with Peripherals

TriMedia

Version 2.1

# Table of Contents

# Chapter 4

# Programming TriMedia Video Applications

## Introduction

This chapter describes how to write video applications using several programming interfaces available on TriMedia.

This chapter begins by describing the high level interface to the Video-In and Video-Out peripherals. These interfaces are provided using the Video Digitizer and Video Renderer components and enable an application to be written without requiring knowledge of the underlying hardware peripherals. An overview of the operation of the TriMedia Video-In and Video-Out units is then presented. This provides background material which is useful when understanding the use of the low-level Video-In/Video-Out device libraries which will then be described.

## TSSA Video Modules

The high level interface is supported using modules which conform to the TriMedia Streaming Software Architecture (TSSA). This software architecture is documented in *Book 3, Software Architecture*. There are several TSSA compliant modules which support video data; examples of interest include the Video Digitizer, the Video Renderer, and the Video Transformer. The Video Digitizer and Video Renderer will be discussed in this chapter, while the Video Transformer is discussed in the next chapter.

### The Video Digitizer

The video digitizer supports video capture using data streaming (*pull* mode) operation; it is described in more detail in Chapter 3, *Video Digitizer (VdigVI) API,* of Book 7, *Video Support Libraries*. In the *pull* mode of operation, the component obtains an empty packet using the datain callback function from an operating system message queue (the *empty* queue). It then captures a video frame, and using the same dataout callback function, places the full packet onto another message queue (the *full* queue). This streaming operation is supported in both the AL and OL layers; the AL API layer assumes no operating system dependencies, while the OL API layer does.

The application can specify parameters which include the video standard (NTSC, PAL, or SECAM), the adaptor type (CVBS or SVIDEO), and the size of the frame to capture.

The exolVrendVO example demonstrates how the Video Digitizer can be used to capture video data. This example will be described in detail after the reader has been introduced to the Video Renderer component.

### The Video Renderer

The VrendVO Video Renderer component is used to display video images using the TriMedia Video-Out peripheral. The component supports non-streaming (*push* mode) and streaming (*pull* mode) operation in the AL Layer. It also supports streaming operation in the OL Layer. In *push* mode, the application calls a Video Renderer function which will display the frame, i.e. the application *pushes* the frame to the renderer.

The Video Renderer supports several video standards and adaptor types. It is also capable of combining the main video with an overlay image with alpha blending. The exolVrendVO example shows the use of this component and will be described next.

### The exolVrendVO Example Program

The exolVrendVO example demonstrates the use of the OL Layer Video Digitizer and Video Renderer. As it uses OL versions of the APIs, data streaming is used to transfer data packets between components. The example simply connects an instance of the Video Digitizer to an instance of the Video Renderer. The digitizer captures live data using the

Video-In device while the renderer displays these images using the Video-Out device. The example enables the user to specify parameters such as the video standard (NTSC or PAL), the adaptor type (CVBS or SVIDEO), and whether to use full resolution or SIF resolution images.

The source code for this example is contained within the examples/exolVrendVO directory of the application tree. This example will now be described in detail.

## Include Files

```
#include <tm1/tmAvFormats.h>
#include "tmos.h"
#include "tmolVrendVO.h"
#include "tmolVdigVI.h"
#include <stdio.h>
#include <tmlib/dprintf.h> /* for debugging with DP(()) */
#include "sys_conf.h"
```

The tmAvFormats.h file contains definitions for the packets which are used to store the video data. The tmos.h file abstracts the underlying operating system; this enables the code to be ported to different operating systems by simply changing this file. The type definitions and function prototypes for the two video components are defined in the tmolVrendVO.h and tmolVdigVI.h files respectively.

## Definitions

```
#define IMAGE_NTSC_HEIGHT 480
#define IMAGE_PAL_HEIGHT  576
#define IMAGE_WIDTH       720
#define IMAGE_STRIDE      768
#define NUM_PACKETS         4
#define NEW_PARAMETER       0
```

The size of the captured and displayed video frame height is defined. Note that the **IMAGE_STRIDE** is larger than the **IMAGE_WIDTH**. This is because the stride must be a multiple of 64 bytes; as the image width is 720 bytes, the nearest 64 byte multiple which is greater than or equal to this is 768.

**NUM_PACKETS** defines the number of packets which will be used to transfer data between the two components. The **NEW_PARAMETER** is used in the user interface code to determine if a new command should be processed.

## Specifying the Packet Format

```
static tmVideoFormat_t digitizer_format = {
   sizeof(tmVideoFormat_t), /* size           */
   0,                       /* hash           */
   0,                       /* referenceCount */
   avdcVideo,               /* dataClass      */
   vtfYUV,                  /* dataType       */
   vdfYUV422Planar,         /* dataSubtype    */
   vdfInterlaced,           /* description    */
   IMAGE_WIDTH,             /* imageWidth     */
   IMAGE_NTSC_HEIGHT,       /* imageHeight    */
   IMAGE_STRIDE             /* imageStride    */
};
```

This structure defines the format of the data contained in the packets. The **hash** and **referenceCount** fields must be set to zero, and should never be modified by the application. They are used by the format manager which ensures that connected components are compatible.

The **dataClass** and **dataType** must always be set to **avdcVideo** and **vtfYUV**. These specify that the class of data is video and is YUV. The **dataSubtype** is set to **vdfYUV422Planar** and specifies the sub-type of YUV data. The Video Digitizer can capture either **vdfYUV422Planar** or **vdfYUV422Interspersed** video; both types store the Y, U, and V components in separate buffers. The **vdfYUV422Planar** sub-type has the chrominance samples co-sited with the luminance data, while **vdfYUV422Interspersed** has the chrominance located mid-way between luminance samples.

The **description** field is set to vdfInterlaced to indicate that the Video Digitizer is capturing interlaced video. The digitizer will store the two fields in a single buffer, with the top field being on the even lines.

Finally, the size of the video frame is specified.

## Static Parameters and Function Prototypes

```
static tmVideoAnalogStandard_t vidStd     = vasNTSC;
static tmVideoAnalogAdapter_t  vidAdapter = vaaCVBS;

extern int __argc;
extern char **__argv;

/* ------ function prototypes  ------ */
static int DoCommand ( char *command );
static int CheckArgcv(int argc, char **argv);
static void PrintUsage(void);

/* setup parameters */
int  vResolution   = viFULLRES;
int  acquStartX    = 0;
int  acquStartY    = 0;
int  scaleUp       = False;
int  voStartX      = 0;
int  voStartY      = 0;
```

The global variables used for the video configuration are declared and initialized. These are used to enable the user to change the configuration by entering commands on the console. The default settings for the video standard and adaptor are NTSC and CVBS. The digitizer will capture full resolution images, and the renderer will perform no upscaling on the output.

The function prototypes are for the user interface code. This will not be described.

## The Main Program

The following code is contained within the **main** function.

### Variables

```
tmLibappErr_t                rval;
Int                          digitizerInstance;
Int                          vrendInstance;
char                         ins[80];
ptmolVdigVICapabilities_t    digCap;
ptmolVrendVOCapabilities_t   rendCap;
ptmolVrendVOInstanceSetup_t  vrend_inst_setup;
ptmolVdigVIInstanceSetup_t   digitizer_inst_setup;
ptsaInOutDescriptorSetup_t   iodSetup;
ptsaInOutDescriptor_t        iod;
```

The **rval** variable is used to store the value returned whenever a call is made to the Video Digitizer, Video Renderer, or tsaDefaults API. The returned value is always of type **tmLibappErr_t** and will have a value of **TMLIBAPP_OK** if there is no error. It is important to check the returned value whenever a call to a component API is made.

The **digitizerInstance** and **vrendInstance** variables are used to store the instance id's when the digitizer and renderer instances are opened. These id's are unique and must be used whenever the application calls a component API function.

The **ins[80]** character array is used to store user command typed in at the keyboard.

Before two components are connected together to form a data flow, the application uses the format manager to determine if they are compatible. Each component has a capabilities structure which specifies what formats it can understand. The **digCap** and **rendCap** variables are used to point to these capabilities structures.

Each component must also be setup before it is used. The **vrend_inst_setup** and **digitizer_inst_setup** are pointers to the instance setup structures.

The connection between two component instances is described using a **tsaInOutDescriptor**. When a descriptor is created, it requires a setup structure which specifies information about the two components being connected and the packets which will be used. The **iodSetup** variable is used to point to this setup information.

## DP Debug Information

```
DPmode(DP_PERSIST);
DPsize(1024*1024);
```

The TriMedia SDE provides a mechanism where debug information can be written to SDRAM by the application and component libraries. This can then be read either during execution if the debugger is being used, or after the program has completed.

## Check Capabilities

```
rval = tmolVdigVIGetCapabilities(&digCap);
rval = tmolVrendVOGetCapabilities(&rendCap);

printf("TriMedia OS Video Renderer Demo. v1.0\n");
printf("\nThis program uses the video digitizer v%d.%d.%d\nand video renderer
v%d.%d.%d\n",
digCap->defaultCapabilities->version.majorVersion,
digCap->defaultCapabilities->version.minorVersion,
digCap->defaultCapabilities->version.buildVersion,
rendCap->defaultCapabilities->version.majorVersion,
rendCap->defaultCapabilities->version.minorVersion,
rendCap->defaultCapabilities->version.buildVersion);
printf("to pass video from video-in to video-out.\n");
```

The capabilities of the two components to be connected together are obtained using the respective GetCapabilities functions. This information will be used by the format manager to ensure the two components are compatible. The versions of the two components are printed on the console.

## Read Command Line Parameters

```
tmosInit();
if( CheckArgcv(__argc, __argv) != 0 ) tmosExit(0);
```

The multitasking operating system is initialized and the command line arguments are checked.

### Open the Components

```
rval = tmolVdigVIOpen(&digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);

rval = tmolVrendVOOpen(&vrendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
```

Before a component can be used, it must first be opened using the respective open function. The relevant function will open an instance of the component, and store a unique instance id in the pointer parameter. The application must use the instance id when calling the components API. It is important to check the return value to ensure that an error did not occur during the open operation. For example, the Video Digitizer and Video Renderer only support a single instance to be open, if the application incorrectly tries to open a second instance then the function will return an error.

### Make the Connection Between the Two Components

```
iodSetup=(ptsaInOutDescriptorSetup_t)malloc(sizeof(tsaInOutDescriptorSetup_t)
   + 2*sizeof(UInt32));
iodSetup->format          = (ptmAvFormat_t)&digitizer_format;
iodSetup->flags           = tsaIODescSetupFlagCacheMalloc;
iodSetup->fullQName       = "full";
iodSetup->emptyQName      = "mpty";
iodSetup->queueFlags      = tmosQueueFlagsStandard;
iodSetup->senderCap       = digCap->defaultCapabilities;
iodSetup->receiverCap     = rendCap->defaultCapabilities;
iodSetup->senderIndex     = VDIGVI_MAIN_OUTPUT;
iodSetup->receiverIndex   = VRENDVO_MAIN_INPUT;
iodSetup->packetBase      = 0;
iodSetup->numberOfPackets = NUMPACKETS;
iodSetup->numberOfBuffers = 3;
iodSetup->bufSize[0]      = IMAGE_STRIDE * IMAGE_PAL_HEIGHT;
iodSetup->bufSize[1]      = IMAGE_STRIDE * IMAGE_PAL_HEIGHT / 2;
iodSetup->bufSize[2]      = IMAGE_STRIDE * IMAGE_PAL_HEIGHT / 2;

rval = tsaDefaultInOutDescriptorCreate(&iod, iodSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
```

The dataflow path connecting two components is specified using an InOutDescriptor. Before this descriptor is created, a structure specifying the connection must be initialized with information which describe the capabilities of the two components and information about the packets which will be placed in the queue.

The first step is to create the setup structure using the standard malloc function. The amount of the memory requested is equal to the size of the **tsaInOutDescriptorSetup_t** structure plus the number of buffers per packet minus one. As the packets store YUV data, three buffers are required per packet, so the application needs to add two extra **UInt32** fields to the allocated memory which will be used to store the U and V buffer sizes. By default, the **tsaInOutDescriptorSetup_t** has space for one buffer size.

The format of the packets which will be placed in the full queue are specified by passing the address of the **digitizer_format** structure. This information is used by the format manager to check that the components can accept this type of packet. It will also be placed automatically on packets when the sender instance (the Video Digitizer in this case) places a packet onto the full queue.

The **flags** parameter is set to **tsaIODescSetupFlagCacheMalloc**. This indicates to the **tsaDefaultInOutDescriptorCreate** function that the packet buffers which it creates must be cache aligned.

Information concerning the queues which will be automatically created are then initialized. The full and empty queues are given names which can be used during debugging; any four letter name can be used. The **queueFlags** parameter provides information which will be used when the full and empty queues are created. The **tmosQueueFlagsStandard** specifies that the queues will be local to the processor (i.e. they do not connect processors) and there is no limit to the number of messages which can be placed on them.

The capabilities of the two components which will be connected together will be checked by the format manager to ensure that they are compatible. The **senderCap** is set to the address of the digitizer capabilities, while the **receiverCap** is set to the renderer capabilities. The component capabilities were obtained previously using the **tmolVdigVIGetCapabilities** and **tmolVrendVOGetCapabilities** functions.

The **senderIndex** and **receiverIndex** fields specify the output and input pins which will be used for the connection. Each component instance uses input and/or output pins for communication to neighboring component instances; each pin represents the full/empty message queue where packets are exchanged. The Video Digitizer has a single output pin referenced by the index value **VDIGVI_MAIN_OUTPUT**. The Video Renderer has two input pins, one for the main video input (**VRENDVO_MAIN_INPUT**) and one for the overlay input (**VRENDVO_OVERLAY_INPUT**). The **receiverIndex** is set to **VRENDVO_MAIN_INPUT** as this pin will receive the video packets for display.

The next set of fields will be used to provide information about the packets which will be automatically created. The **packetBase** field is used to specify an identification number to the packets that are placed in the queues. The application can use any number; the first packet will contain this value, with subsequent packets containing id's with ascending values. In the example, the first packet will have an id of zero, the second packet will be one, the third will be two, and the forth packet will have an id of three. This can be useful for debugging to identify where the packets are being held. The **numberOfPackets** specifies the number of packets which must be created and stored in the empty queue. The **numberOfBuffers** specifies the number of data buffers per packet. As the components are using YUV data, three buffers are required per packet to store the Y, U, and V data. Each buffer has a corresponding **buffSize** value which specifies the size of the buffer. As the packets are hold YUV data, the first **bufferSize** is set to the size of the luminance component, with the subsequent **bufferSize** values set to the size of the Chrominance components. As the data is YUV422, the chrominance is half the size of the luminance.

Finally, the InOutDescriptor is created using **tsaDefaultInOutDescriptorCreate**. This creates the descriptor, the message queues, and the associated packets.

## Set Up the Video Digitizer and Renderer

```
rval = tmolVdigVIGetInstanceSetup(digitizerInstance,&digitizer_inst_setup);
tmAssert((rval == TMLIBAPP_OK), rval);

rval = tmolVrendVOGetInstanceSetup(vrendInstance,&vrend_inst_setup);
tmAssert((rval == TMLIBAPP_OK), rval);

digitizer_inst_setup->instSetup>outputDescriptors[VDIGVI_MAIN_OUTPUT] = iod;
digitizer_inst_setup->videoStandard = vidStd;
digitizer_inst_setup->videoAdapter  = vidAdapter;
digitizer_inst_setup->capSizeFlag   = vResolution;
digitizer_inst_setup->startX        = acquStartX;
digitizer_inst_setup->startY        = acquStartY;

vrend_inst_setup->instSetup->inputDescriptors[VRENDVO_MAIN_INPUT] = iod;
vrend_inst_setup->videoStandard   = vidStd;
vrend_inst_setup->adapterType     = vidAdapter;
vrend_inst_setup->scaleUp         = scaleUp;
vrend_inst_setup->imageHorzOffset = voStartX;
vrend_inst_setup->imageVertOffset = voStartY;

rval = tmolVdigVIInstanceSetup(digitizerInstance, digitizer_inst_setup);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("digitizer initialized.\n");

rval = tmolVrendVOInstanceSetup(vrendInstance, vrend_inst_setup);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("renderer initialized.\n");
```

Before an instance of a component is used it must first be setup. The first step is to obtain a pointer to the instance setup structure; this is achieved by calling the **tmolVdigVIGetInstanceSetup** and **tmolVrendVOGetInstanceSetup** respectively.

The Video Digitizer structure is setup first. The instances output descriptor is set to point to the InOutDescriptor which was created in the last section of code. The input **videoStandard** specifies either PAL or NTSC; by default, this is set to NTSC. The input **videoAdaptor** indicates the adaptor type and can be CVBS or SVIDEO, with the CVBS being set by default. The **capSizeFlag** indicates whether to perform full resolution or half resolution video capture; by default this will be full resolution. Finally, the **startX** and **startY** fields are used to specify the location in the incoming field where video capture will start. The two values are zero by default.

The Video Renderer parameters are then initialized. The instances main image input descriptor is set to the InOutDescriptor which was created before. The output video standard and adaptor type are setup in similar fashion to the Video Digitizer. The **scaleUp** flag is used to specify that the input image should be scaled up by the video-out hardware. If full resolution images are captured, this value should be set to false. Half resolution images may be scaled up to full resolution by setting this value to true. Finally, the **imageHorzOffset** and **imageVertOffset** specify the starting pixel and line in the active output video area where the image will be displayed. These are set to zero by default.

Once the setup structures have been initialized, the **tmolVdigVIInstanceSetup** and **tmolVrendVoInstanceSetup** functions are called to pass the information to the two instances.

### Starting the Component Instances

```
DP(("\nStarting Video Renderer\n"));
rval = tmolVrendVOStart(vrendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("renderer started.\n");

DP(("\nStarting Video Digitizer\n"));
rval = tmolVdigVIStart(digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("digitizer started.\n");
```

Data streaming between the two component instances will begin once both have been started. The **tmolVdigVIStart** and **tmolVrendVOStart** functions will initiated data streaming for each instance. Both components execute in interrupt service routines.

## User Input

```
printf(
"\nVideo Renderer demo started.\nVideo input echoed to video output.\n");
PrintUsage();
printf("Enter Command:\n");
while (1){
  printf(">");
  gets(ins);
  rval = DoCommand(ins);
  if( rval == NEW_PARAMETER ) {
    if( rval = tmolVdigVIStop(digitizerInstance) )
      printf("exolVrendVO: tmolVdigVIStop error %s\n",rval);
    if( rval = tmolVrendVOStop(vrendInstance) )
      printf("exolVrendVO: tmolVrendVOStop error %s\n",rval);
    digitizer_inst_setup->videoStandard = vidStd;
    digitizer_inst_setup->videoAdapter  = vidAdapter;
    digitizer_inst_setup->capSizeFlag   = vResolution;
    digitizer_inst_setup->startX        = acquStartX;
    digitizer_inst_setup->startY        = acquStartY;
    vrend_inst_setup->videoStandard     = vidStd;
    vrend_inst_setup->adapterType       = vidAdapter;
    vrend_inst_setup->scaleUp           = scaleUp;
    vrend_inst_setup->imageHorzOffset   = voStartX;
    vrend_inst_setup->imageVertOffset   = voStartY;
    tsaDefaultInstallFormat(iod,(ptmAvFormat_t)&digitizer_format);
    if( rval = tmolVdigVIInstanceSetup(digitizerInstance,
                                   digitizer_inst_setup) )
      printf("exolVrendVO: tmolVdigVIInstanceSetup error %s\n", rval);
    if( rval = tmolVrendVOInstanceSetup(vrendInstance,
                                    vrend_inst_setup) )
      printf("exolVrendVO: tmolVrendVOInstanceSetup error %s\n",rval);
    if( rval = tmolVrendVOStart(vrendInstance) )
      printf("exolVrendVO: tmolVrendVOStart error %s\n",rval);
    if( rval = tmolVdigVIStart(digitizerInstance) )
      printf("exolVrendVO: tmolVdigVIStart error %s\n",rval);
  }else if (rval == -1){
    break;
  }
}
```

The example enables the user to enter commands via the console which alter the digitizer and renderer parameters. while the two video instances are streaming data, the default task will wait for the user to type a command. Once a valid command has been entered, the two instances are stopped by calling **tmolVdigVIStop** and **tmolVrendVOStop** respectively; this will terminate data streaming. The new instance values are then assigned to the respective instance setup structures and each component instance is setup. Finally, data streaming is restarted for both the digitizer and renderer.

If the user types ‘exit’ at the console, then the ‘while’ processing loop with be exited, and the shutdown sequence of command will be executed.

### Stop and Shutdown

```
printf("\nStopping Everything:\n");
DP(("\nStopping Everything:\n"));
rval = tmolVdigVIStop(digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVrendVOStop(vrendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVdigVIClose(digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVrendVOClose(vrendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);

rval = tsaDefaultCheckQueues(iod);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("tsaDefaultCheckQueues returned 0x%x\n", rval);

printf("Destroying InOutDescriptor\n");
rval = tsaDefaultInOutDescriptorDestroy(iod);
tmAssert((rval == TMLIBAPP_OK), rval);

DP(("Demo Complete.\n"));
printf("Demo Complete.  \n");
tmosExit(0);
```

Component instances should be stopped before they are closed. The **tmolVdigVIStop** and **tmolVrendVOStop** functions will cause the two instances to stop data streaming and return any packets that they may be holding. The low level video-in and video-out devices will also be stopped within these functions.

After use, each component instance should be closed. For the Video Digitizer and Renderer which only allow one instance to be opened, this will enable other applications or tasks to use the components. The **tmolVdigVOClose** and **tmolVrendVOClose** will free any memory that was being used by the instances.

The InOutDescriptor full and empty queues can be checked using the **tsaDefaultCheck-Queues** function. This function should be used during debugging and checks the queues to ensure that the correct number of packets have been returned to them.

Finally, the InOutDescriptor should be destroyed by calling **tsaDefaultInOutDescrip-torDestroy.** This will remove the packets contained within the queues, free the memory allocated to the packets, and free the memory allocated to the InOutDescriptor.

## TriMedia Video-In Operation

The TriMedia Video-In unit provides digital video input in YUV 4:2:2 with 8-bit resolution, multiplexed in CCIR656 format from a digital camera or CCIR656-capable video decoder (such as a Philips SAA7111), across an 8-bit wide interface.

The Video-In unit can operate in any one of the following modes:

- Full-resolution capture
- Half-resolution capture
- Raw capture (raw8, raw10s, and raw10u)
- Message passing

An operation in each of these modes is given below. For more information, see the data book of the appropriate TriMedia.

### Full-Resolution Capture Mode

In Full-resolution Capture mode, the Video-In unit receives all three video components (Y, U, and V), as well as synchronization information, on the 8-bit wide interface in CCIR656 format. The Y, U, and V video components are separated into three different streams. Each component is written in packed form into Y, U, and V buffers in the SDRAM. This is commonly called a *planar format*.

The DSPCPU initiates capture by setting the **CAPTURE_ENABLE** flag to **1**. The Video-In unit captures video data and stores it in the SDRAM, at the locations defined by the storage parameters defined in the MMIO registers. When capture is complete (that is, any internal Video-In buffers have been flushed and the entire captured image is in local SDRAM), Video-In sets the **STATUS** register flag to **CAPTURE_COMPLETE**. This causes a DSPCPU interrupt to be requested. The Video-In unit resumes capture as soon as the DSPCPU acknowledges the previously captured image by deactivating **CAPTURE_COMPLETE**.

You can program the **Y_THRESHOLD** field to generate pre-completion (or post-completion) interrupts. Whenever **CUR_Y** reaches the **Y_THRESHOLD**, the **THRESHOLD_FLAG** in the status register is set. If enabled in the Video-In control register, this event causes a DSPCPU interrupt request.

If the Video-In internal buffers overflow because of insufficient internal data-highway bandwidth allocation, the **HIGHWAY_BANDWIDTH_ERROR** condition is raised in the Video-In status register (**VI_STATUS**). If enabled, this causes a DSPCPU interrupt request. Capture continues at the correct memory address as soon as the internal buffers can be written to memory, but one or more pixels may be lost, and the corresponding memory locations are not written.

## Full-Resolution Capture Mode

Full-Resolution Capture mode is illustrated in the vivot example that follows.

## Half-Resolution Capture Mode

Half-Resolution Capture mode is identical in operation to full-resolution capture mode, except that horizontal resolution is reduced by a factor of 2 on both luminance and chrominance data.
Half-Resolution Capture mode is used for CIF format in the vivot example that follows.

## Raw Capture Mode

All Raw Capture modes (raw8, raw10s, and raw10u) behave similarly. The video data is captured at the rate of the sender's clock, without interpretation or start/stop on the basis of the data values.

The DSPCPU initiates capture by providing two empty buffers and putting their base addresses and sizes in the **BASEn** and **SIZEn** registers. It does so by writing a base address and size to MMIO control fields. After two buffers are assigned, capture is enabled by setting **CAPTURE_ENABLE** to **1**. The Video-In unit starts capturing video data in **buffer1** (the active buffer). It continues until capture is disabled or **buffer1** fills up. If **buffer1** fills up, capture continues (without missing a sample) in **buffer2**. At the same time, **BUF1FULL** is asserted, which causes an interrupt on the DSPCPU.

**buffer2** then becomes the active buffer and the loop repeats. In normal operation, the DSPCPU before **buffer2** fills up, the DSPCPU must assign a new, empty buffer **BASE1**, **SIZE1** and perform an **ACK1** operation. If the DSPCPU fails to assign a new **buffer1** before **buffer2** fills up, the **OVERRUN** condition is raised, bringing a temporary halt to capture. Capture resumes as soon as the DSPCPU makes one or more new buffers available through an **ACK1** or **ACK2** operation.

If insufficient bandwidth is allocated from the internal data highway, the Video-In internal buffers might overflow. This leads to assertion of the **HIGHWAY_BANDWIDTH_ERROR** condition. One of more data samples are lost. Capture resumes at the correct memory address as soon as the internal buffer is written to memory.

## Message-Passing Mode

In Message-Passing mode, the Video-In unit receives 8-bit message data across the 8-bit wide interface. It writes the message data in packed form (four 8-bit message bytes per 32-bit word) to the SDRAM. Message data capture starts on receipt of a **START** event and continues until either the receive buffer is full, or the **EndOfMessage** event is received. **OVERFLOW** is raised if a receive buffer is full and no **EndOfMessage** event has been received. If enabled, it generates a DSPCPU interrupt. Detection of overflow leads to total halt of capture of this message. Capture resumes in the next buffer on receipt of the next **START** event.

The TriMedia Video-In APIs provide the necessary interface for video applications to access the TriMedia Video-In unit hardware.

## TriMedia Video-Out Operation

The TriMedia Video-Out unit connects to an off-chip video subsystem, such as a digital video encoder chip (DENC), a digital video recorder, or the video input of another TriMedia system through a CCIR656-compatible byte-parallel video interface.

The Video-Out unit outputs digital video in YUV 4:2:2 co-sited format with 8-bit resolution multiplexed in CCIR656 format. It can drive a CCIR656-compatible digital video encoder across an 8-bit wide interface. It can also drive other CCIR656-compatible devices, such as digital video cassette recorders (VCRs) and the Video-In unit of other TriMedia chips. For example, in Video-In Diagnostic Mode, the Video-Out unit of one TriMedia supplies video data to the Video-In unit of a second TriMedia system.

The Video-Out unit can operate in either *image transfer* or *data transfer* (data streaming or message-passing) mode. The TriMedia DSPCPU programs the Video-Out unit by setting the **Mode** field to the appropriate transfer mode, setting the appropriate addresses, address deltas, image-timing registers, and associated control bits in the control register. Setting **VO_ENABLE** in the **VO_CNTRL** register starts the Video-Out unit, which transfers the image or messages as commanded.

In image-transfer and data-streaming modes, the Video-Out unit runs continuously. It issues an interrupt to the DSPCPU at the end of each field. To maintain continuous video output, the DSPCPU updates the Video-Out image data pointers with pointers to the next field during the vertical blanking interval. In message-passing mode, the Video-Out unit runs until the message has been transferred.

### Image Transfer Mode

In Image Transfer mode, the Video-Out unit continuously transfers an image from the SDRAM to the Video-Out port. The mode field in the **VO_CTL** register defines the image input data format and whether or not the Video-Out unit is to perform horizontal upscaling. The Video-Out unit accepts memory image data in YUV 4:2:2 co-sited, YUV 4:2:2 interspersed, and YUV 4:2:0 co-sited image output streams.

During image transfer, the YTR bits are set in the status register when the Image Line Counter reaches the **Y_THRESHOLD** value. When an image field has been transferred, the **BFR1_EMPTY** bit is set in the status register. The DSPCPU is interrupted when either the YTR or the **BFR1_EMPTY** flag is set and its corresponding interrupt is enabled.

To maintain continuous transfer of image fields, the DSPCPU supplies new pointers for the field following each **BFR1_EMPTY** interrupt. If the DSPCPU does not supply new pointers before the next field, the **URUN** bit is set, and the Video-Out unit uses the same pointer values until they are updated.

Image Transfer mode is illustrated in the vivot example that follows.

## Data Transfer Modes

There are two modes for transferring data:

- Data-Streaming mode
- Message-Passing mode

### Data-Streaming Mode

In the Data-Streaming mode, the Video-Out unit generates a continuous stream of byte data using internal or external clocking. Dual buffers facilitate continuous data streaming by allowing the DSPCPU to set up the next buffer while the first one is being emptied by the Video-Out unit.

The data is stored in the DRAM in two buffer tables. When the Video-Out unit has transferred the contents of one table, it interrupts the DSPCPU and begins transferring the contents of the second table. The DSPCPU supplies pointers to both tables. The Video-Out unit supplies a continuous stream of data to the video device, provided the DSPCPU updates the pointer to the next table before the Video-Out starts transferring data from the next table.

When each buffer has been transferred, the corresponding buffer empty bit is set in the status register. The DSPCPU is interrupted if the buffer empty interrupt is enabled. To maintain continuous transfer of data, the DSPCPU supplies new pointers for the next data buffer following each buffer empty interrupt. If the DSPCPU does not supply new pointers before the next field, the Video-Out unit uses the same pointer values until they are updated.

### Message-Passing Mode

In the Message-Passing mode, messages can be sent to one or more TriMedia Video-In units. Start and end-of-message signals are provided in this mode to synchronize message passing to the other TriMedia message receivers. Video data is stored in the DRAM in one buffer table.

Setting **VO_ENABLE** in the **VO_CNTRL** register starts the Video-Out in Message-Passing mode. The Video-Out unit sends a **Start** condition on **VO_IO1**. When the Video-Out unit has transferred the contents of the buffer table, it sends an **End** condition on **VO_IO2**, sets the **BFR1_EMPTY** bit, and interrupts the DSPCPU. The Video-Out unit stops. No further operation takes place until the DSPCPU sets **VO_ENABLE** for another message or another Video-Out operation.

The TriMedia Video-Out APIs provide the necessary interface for video applications to access the TriMedia Video-Out unit hardware.

## Using the TriMedia Video-In/Video-Out Device Library

The APIs provided in the TriMedia Video-In/Video-Out Device Library enable you to access both the Video-In and Video-Out hardware units of TriMedia. The Video-In/Video-Out device library provides functions for controlling video encoders and decoders. It can be linked with other programs, providing you with total control of the hardware by enabling you to

- Optimize ISRs to meet application requirements.
- Create vendor-specific initialization and configuration routines for on-board chips (such as a decoder that works with the TriMedia Video-In component and an encoder that works with the TriMedia Video-Out component).

### Guidelines for Use of the Video-In/Video-Out APIs

General guidelines for using the TriMedia Video-In/Video-Out APIs are as follows:

- Use the archive version (libdev.a), rather than building the library yourself. (The Video-In/Video-Out device library is archived in libdev.a).
- The source for the Video-In/ Video-Out device library is included in the TCS. This makes it easier to incorporate new versions of the library as they become available.
- Pass the specific instance when making subsequent calls.
- The Video-In/Video-Out Device Library operates as an exclusive device driver, and, as such, can service only one task at a time. This is enforced through the instance identifier, which is returned by all the initialization functions.
- Modify the functions, **viOpen** and **voOpen** using interfaces provided in the Board Support API.

The **viOpen** and **voOpen** functions call the initialization routines for the analog I/O hardware on the board. The board library provides support for default boards (For example, the TriMedia debug board and IREF board).

It provides the initialization routine for the decoder on the debug board (SAA7111) and IREF board (SAA7111A), and for the encoder on the debug board (SAA7185) and IREF board (SAA7125).

For more information about the Board Support API, refer to Reference Manual II of the Philips TriMedia SDE.

- Check the error values returned by the initialization functions. Most of the Video library functions return zero on success, or nonzero error codes.
- Use the debug version of the libdev.a library during development. Many functions check and report the use of sizes and alignments that the hardware cannot support.

## Vivot Demonstration Program Overview

The vivot example is intended as an example to allow the user to gain familiarity with the techniques necessary to program the video capabilities of the TriMedia architecture. The program demonstrates how the video input and output modes can be reprogrammed dynamically on TriMedia.

*First*, the image captured on VI is converted to CIF and a quarter sized image on the middle of the screen (**vivoRunCIF**).

*Next*, a full screen image is displayed and captured (**vivoRunFullres**). These two processes are executed for 1000 frames each.

*Finally*, a quarter sized image is displayed on the middle of the screen on top of the TriMedia logo (**vivoRunOverlay**), thus illustrating the overlay feature.

It is useful to know a certain number of “video programming tricks” when using TriMedia; for example, CIF conversion, as well as general device issues on TriMedia (such as buffer alignment, or cache coherency). The data book provides the functionalities but it does not go into detail. The purpose of this section is to explain these functionalities.

The programmer needs to be aware of a certain number of implementation choices while studying the code. For example, the buffer scheme being used indexes buffers from a table and references them circularly using modulo addressing. Using a linked list of buffers can be preferable. Optimizing the code could reduce the buffer space requirements or execution time. For example, a single buffer can be used for Video-In (VI) and Video-Out (VO). This was not done for clarity reasons. (For example, the overlay buffer is used for the captured frames and the VO buffer for the logo). Busy waiting is used for buffer and ICP processing, instead of a semaphore.

### C Program Includes

Most C programs include **<stdio.h>** and **<stdlib.h>**. TriMedia-specific C library functions are in **<tmlib/tmlibc.h>**. The standard C header files such as **<assert.h>** and **<ctype.h>** can be included also. MMIO registers are defined in **<tm1/mmio.h>**.

Custom ops are defined in **<ops/custom_defs.h>**. Multimedia formats (such as **vaaNTSC**, **vaaPAL**, **vaaCVBS**, **vaaSvideo**) are defined in **<tm1/tmAvFormats.h>**.

A program that uses VO should include **<tm1/tmVO.h>** and **<tm1/tmVOmmio.h>**. To use the Image Co-Processor (ICP), include **<tm1/tmICP.h>**. A program that uses VI should include **<tm1/tmVI.h>** and **<tm1/tmVImmio.h>**.

To find out the clock speed or the processor type, include **<tm1/tmProcessor.h>** (**procGetCapabilities**) and/or **<tm1/tmBoard.h>**.

For definitions associated with interrupts, include **<tm1/tmInterrupts.h>**. To use the DP debug printing facility of **tmgmon**, include **<tmlib/dprintf.h>**.

## Main Program

The first line initializes the **DP** printing facility of **tmgmon**. The next two lines print a start-up message, using **DP** and **printf**. The call to **Reportsys** is to find out the processor clock frequency and the version of the processor (for work-arounds).

Video out bug 21727 was present in versions of the processor prior to TM1S1.1. Video out bug 3056 is less important but the two exacerbate each other. The call to **vivoDetect-workarounds** detects which of these bugs are present and positions the flag **DummyCode**. We will assume in what follows that **DummyCode** is zero.

The call to **vivoCheckArgcV** sets the adapter type (S-video, composite) and the video standard (PAL or NTSC). The call to **vivoRun** contains the main program.

```
int
main(int argc, char **argv){
   SetDP();
   DP((Header));
   printf(Header);
   reportSys();
   vivoDetectworkarounds();

#ifndef __TCS_nohost__
   vivoCheckArgcv(argc, argv);
#endif

   vivoRun();
   exit(0);
}
```

## Vivot Demonstration Program (Vivorun)

- Buffer allocation is ensured by **vivoAlloc**.
- **vivoOpenAPI** calls **viOpenAPI** and **voOpenAPI** for API initialization.
- **vivoCloseAPI** frees the buffers (the name is a misnomer).

The code for **vivoRun** is shown below.

```
vivoRun(){
   vivoAlloc();
   vivoOpenAPI();

   vivoRunCIF();
   vivoRunFullRes();
   vivoRunOverlay();
   vivoCloseAPI();
}
```

### Image Representation

The output format is defined by the width, the height, and the stride. The stride is different from the width because lines need not be contiguous and because of alignment of lines to cache boundaries.

The dimensions are 352 x 240 for CIF (**cifWidth**, **cifHeight**) and 720 x 480 (full Width, full Height) for full resolution. The image buffer sizes are 384 x 240 for CIF and 768 x 576 for full resolution.

The strides (**cifStride**, **fullStride**) differ from the widths because VI requires that image lines begin on a cache line (modulo 64 bytes) boundary. The image is represented in planar format using separate Y, U, and V pointers in the **vbuf** structure.

## Buffer Allocation (vivoAlloc)

**VivoAlloc** calls **allocCif422**, **allocFullres**, and **allocBkBuf** to allocate the CIF, full resolution, and overlay buffers.

Table 1 summarizes the buffer allocation in the demonstration program.

- The buffer allocation scheme is fixed and there is no sharing.
- Buffers are addressed via an index modulo the total number (4).
- The buffers are circulated between VI, VO and processing.
- Pointer advancement corresponds to inputs and completion of processing.
- The flag field of the vbuf structure identifies the state at any given point in time (**VID_RDY_VI**, **VID_RDY_VO**, **VID_RDY_MM**).
- 5 megabytes of memory are required for the buffers in total.
- The dimensions in full resolution are large enough to contain either a PAL (704×576) or NTSC (720×480) image.

**Table 1** Buffer Allocation in Demonstration Program

| | | | | | |
|---|---|---|---|---|---|
| CIF | 384 | 240 | 2 | 4 | 737280 |
| full res | 768 | 576 | 2 | 4 | 3538944 |
| overlay | 768 | 576 | 2 | 4 | 884736 |
| Total | | | | | 5160960 |

## Cache Management

Cache coherency between the DSPCPU and the peripheral units is managed in software on the TriMedia processor. The program contains routines to allocate a buffer, to update the cache to memory, and to remove stale data.

These routines deal with cache lines (blocks whose sizes are a multiple of 64 bytes beginning at a modulo 64 boundary). **allocSz** calls the library routine **_cache_malloc** (Refer to code insert below).

The second parameter indicates the set number from which to allocate (0-31 or -1 if any is acceptable). Refer to Chapter 3 of the *Cookbook* for information about the incidence of this on performance.

The pointer returned by **_cache_malloc** begins at a modulo 64 boundary. The size is rounded up also.

```
UInt32
allocSz(int bufSz){
   UInt32 temp;
   int i;

   if((temp=(UInt32)_cache_malloc(bufSz,-1))==Null)
      my_abort("_cache_malloc", 0);
   memset(temp, 0, bufSz);
   _cache_copyback(temp, bufSz);
   return temp;
}
```

### viOpenAPI - level 1 initialization for VI

The code for **viOpenAPI** is shown in two sections below. The call to **viOpen** acquires the peripheral; accesses to a peripheral have to be exclusive. This returns an "instance" in **viInst** corresponding to the peripheral.

```
void viOpenAPI(){
   tmLibdevErr_t err;

   if( err = viOpen(&viInst) ) my_abort("viOpen", err);
```

The call to **viInstanceSetup** programs the 7111 and associates the interrupt service routine **viTestISR** at interrupt priority level 3 with the device.

```
   memset((char *) (&viInstSup), 0, sizeof (viInstanceSetup_t));

   viInstSup.interruptPriority = intPRIO_3;
   viInstSup.isr = viTestISR;
   viInstSup.videoStandard = videoStandard;
   viInstSup.adapterType = adapterType;

   if( err = viInstanceSetup(viInst,&viInstSup) )
      my_abort("viInstanceSetup", err);
}
```

### voOpenAPI - level 1 initialization for VO

The code for **voOpenAPI** follows the same general structure as **viOpenAPI**. An instance is allocated and then setup.

```
void
voOpenAPI(){
   tmLibdevErr_t err;
   pprocCapabilities_t procCap;

   if (err = voOpen(&voInst)) my_abort("voOpen", err);
   memset((char *) (&voInstSup), 0, sizeof (voInstanceSetup_t));

   voInstSup.interruptPriority = intPRIO_6;
   voInstSup.isr               = voTestISR;
   voInstSup.videoStandard     = videoStandard;
   voInstSup.adapterType       = adapterType;

   procGetCapabilities(&procCap);
/* see formula on VO, Figure 7.6 in the data book */
   voInstSup.ddsFrequency =
   (unsigned int)(0.5 + (1431655765.0*27000000/procCap->cpuClockFrequency));

   voInstSup.hbeEnable      = True;
   voInstSup.underrunEnable = True;

   if (err = voInstanceSetup(voInst, &voInstSup))
   my_abort("voInstanceSetup", err);
}
```

The interrupt service routine is **voTestISR** and the interrupt is at level 6. The Highway Bandwidth Error (HBE) interrupt is enabled. This corresponds to VO not getting data from the highway in time to continue transfer. The **Underrun** interrupt is enabled. This corresponds to the CPU not updating the buffer pointer in time (excessive interrupt latency). For more information on these, refer to section 7.12.3 of the data book.

The initialization of the DDS frequency merits some explanation. Section 7.4 of the data book defines the clock frequency at the output by the following equation:

$$f_{DDS} = \frac{3 \times FREQUENCY \times f_{DSPCPUCLK}}{2^{32}}$$

The value for fdds is twice the video clock frequency of 13.5 Mhz. The input divider for the clock frequency divides by two (see Table 7-7 of the data book, default values for the PLL fields in VO_CTL). The frequency of 27 Mhz corresponds for PAL to an image format of 864 pixels, 625 lines, at a 25 Hz frame rate (50 Hz interlaced). The image format parameters for NTSC vary, but the clock frequency is identical. So the value for fdds needs to be 27 Mhz.

The code in **voOpen** corresponds to a rearrangement of the terms to obtain the **ddsfrequency** of the equation above.

```
voInstSup.ddsFrequency = (unsigned int)
        (0.5+(1431655765.0* 27000000/(float)procCap->cpuClockFrequency));
```

The number 1,431,655,765 (referenced in the code above) equals

$$\frac{2^{32}}{3}$$

## Field Capture versus Frame Capture

The dimensions of image being captured depend on the output resolution. In full resolution, the two fields are assembled together to form a frame. Consecutive lines from different fields are assembled together to form an image by using a stride equal to twice the line stride and setting buffer pointers.

In CIF resolution, the buffer consists of a single field and has half the height of the image. The frame rate for output is the same as in full resolution since one of out two fields is discarded. For the horizontal resolution the HALFRES mode of the Video Out unit is used.

## Running in CIF Resolution (vivoRunCIF)

The code for **vivoRunCIF** begins by initializing the capture buffer pointers.

```
void
vivoRunCIF(){
   tmLibdevErr_t err;

   printf("\nStarting CIF resolution mode\n");

   cpGenBuf(cif422Buf, VID_NUMBUFS, VID_RDY_VI);
   viNum = mmNum = voNum = 0;
```

The resolution for U and V strides are half that of Y so the stride must be divided by two also.

```
yFieldStride = cifStride;
uvFieldStride = (cifStride >> 1);
overlayFieldStride = 0;
```

Setting **capfield** tells **viTestISR** *not* to assemble fields into frames. This has the effect of dividing by two the vertical resolution.

```
capField = True;
firstField = False;
```

The arguments to **viYUVAPI** indicate that the image is being captured starting at line 11, pixel 4, with field capture. The HALFRES mode is used, dividing in effect by two the horizontal resolution.

```
viYUVAPI( viHALFRES, cifWidth, cifHeight, cifStride, 1, 4, 11,
          (Pointer)(cif422Buf[0].Y),
          (Pointer)(cif422Buf[0].U),
          (Pointer)(cif422Buf[0].V) );
```

The arguments to **voYUVAPI** indicate that the image is offset to line 64, pixel 128. The output format is in 4:2:2 format with cosited sampling for luminance and chrominance.

This corresponds to the format used by VI (CCIF 656 standard). The VO unit has the capacity to upscale the image by two but this is not used.

```
voYUVAPI( vo422_COSITED_UNSCALED, cifWidth, cifHeight, cifStride, 64, 128,
          (Pointer)cif422Buf[0].Y,
          (Pointer)cif422Buf[0].U,
          (Pointer)cif422Buf[0].V );
```

1000 frames are copied from VI to VO. This corresponds to approximately 33 seconds at 60 Hz.

```
for (voISRCount = 0; voISRCount < loopCount;) {
   mmBufUpdate();
}
```

After 1000 frames, we shut down image display and capture.

```
if (err = viStop(viInst)) my_abort("viStop", err);
if (err = voStop(voInst)) my_abort("voStop", err);
```

## Running in Full Resolution (vivoRunFullRes)

In full resolution, two fields are assembled to form a frame together. The global variables **ScanWidth** and **uvScanWidth**. These values are used in the ISR to adjust the buffer pointers for the second field of capture. They correspond to the offset in bytes between fields for the Y and U, V buffers.

```
yScanWidth = fullStride;
uvScanWidth = (fullStride >> 1);
```

This corresponds to forming the frame by reassembling consecutive lines from different fields together. The arguments to **viYUVAPI** indicate that the image is being captured starting at line 21, pixel 0, with a frame mode of capture, (as explained previously).

```
viYUVAPI( viFULLRES, fullWidth, fullHeight, fullStride, 0, 0, 21,
          (Pointer)(fullResBuf[0].Y),
          (Pointer)(fullResBuf[0].U),
          (Pointer)(fullResBuf[0].V) );
```

The arguments to **voYUVAPI** correspond to those used in CIF mode except that the image is offset at (0, 0).

```
voYUVAPI( vo422_COSITED_UNSCALED, fullWidth, fullHeight, fullWidth, 0, 0,
          (Pointer)fullResBuf[0].Y,
          (Pointer)fullResBuf[0].U,
          (Pointer)fullResBuf[0].V );
```

## Initialization With Alpha Overlay (vivoRunOverlay)

In overlay mode the TM-1 logo is displayed on video out. The image from video in is converted to overlay mode and output.

```
void vivoRunOverlay(){
    tmLibdevErr_t err;

    printf("\nStarting overlay mode\n");

    cpGenBuf( cif422Buf, VID_NUMBUFS, VID_RDY_VI );
    runningOverlay = True;
    cpUsize = ((cifStride * cifHeight) >> 1);
```

The image is captured starting at line 12 (hex C), pixel 12, with field capture.

```
viYUVAPI( viHALFRES, cifWidth, cifHeight, cifStride, 1,
         0xc,       /* x offset */
         0xc,       /* y offset */
         (Pointer)(cif422Buf[0].Y),
         (Pointer)(cif422Buf[0].U),
         (Pointer)(cif422Buf[0].V) );
```

The output format is the same as in full resolution mode.

```
 voYUVAPI(vo422_COSITED_UNSCALED, fullWidth, fullHeight, fullWidth, 0, 0,
(Pointer)fullResBuf[0].Y,
(Pointer)fullResBuf[0].U,
(Pointer)fullResBuf[0].V);
```

**voOverlayAPI** is then called. The buffer pointer corresponds to the first CIF buffer. In overlay mode, the VO buffer points to the TriMedia logo. The overlaid zone is at offset (64, 128) from the left hand corner of the active video area and has the width and height a CIF image (350 x 240).

The stride value of a single alpha value of 64 (50 percent) is used over the entire display image. The CIF buffer Y pointer points to the converted overlaid image. A single pointer is used as the overlaid image is in YVYU format.

The stride of 1408 is four times the width of the image because in the buffer both the even and odd fields and the luminance and chrominance data are interspersed.

```
voOverlayAPI(64, 128, 352, 120, 64, 64, 1408, (Pointer) cif422Buf[0].Y);
```

1000 buffers are copied from VI to VO.

```
for (voISRCount = 0; voISRCount < loopCount;) {
mmOvlyBufUpdate();
}
```

Image capture and display are stopped as previously.

```
if (err = viStop(viInst)) my_abort("viStop", err);
if (err = voStop(voInst)) my_abort("voStop", err);
```

## Setup Input and Begin Capture (viYUVOpenAPI)

The beginning of the code for **viYUVAPI** is shown below. The VI unit has two interrupt modes corresponding to when a scan line is reached (**thresholdReached**) and to capture complete (end of an image, beginning of vertical sync interval). The capture mode is used. Cosited sampling is used.

```
void
viYUVAPI(int mode, int width, int height, int stride, int fieldBuf,
         int startx, int starty, Pointer yBase, Pointer uBase, Pointer vBase){
   tmLibdevErr_t err;

   memset((char *) (&viYUVSup), 0, sizeof (viYUVSetup_t));

   viYUVSup.thresholdReachedEnable = False;
   viYUVSup.captureCompleteEnable = True;
   viYUVSup.cositedSampling = True;
```

The threshold register is set to line 0. The **startX**, and **startY** values correspond to the line and pixel number to begin image capture. The width parameter corresponds to the number of pixels after the starting pixel for line capture

```
viYUVSup.mode = viFULLRES;
viYUVSup.yThreshold = 0;
viYUVSup.startX = startx;
viYUVSup.startY = starty;
viYUVSup.width = width;
```

The next three instructions initialize the VI's units buffers pointers.

```
viYUVSup.yBase = yBase;
viYUVSup.uBase = (DummyCode) ? (Pointer) MMIO(DRAM_BASE) : uBase;
viYUVSup.vBase = vBase;
```

Depending on whether the captured image is in full resolution, or in CIF mode, it must be assembled from fields to frames by the VI ISR.

The first case corresponds to CIF mode. Interlacing is used in this mode to divide the vertical resolution and the second field is eliminated. The delta values for U and V are divided because they have half the resolution.

```
if (fieldBuf) {
   viYUVSup.height = height;
   viYUVSup.yDelta = (stride - width) + 1;
   viYUVSup.uDelta = ((stride - width) >> 1) + 1;
   viYUVSup.vDelta = ((stride - width) >> 1) + 1;
}
```

"Delta" corresponds to the difference between the last pixel of a line and the first pixel of the following line. This corresponds to the difference between "stride" and "width" (the space necessary so that the next line can begin on a mod 64 boundary).

The "+1" comes from the definition of Delta (the pointer stops incrementing at the last pixel). The second case corresponds to full resolution mode. The value for height corresponds to the number of lines in a field (half that of a full image). The extra space of "stride" bytes corresponds to the corresponding line from the other field of the image.

```
else {
   viYUVSup.height = (height >> 1);
   viYUVSup.yDelta = (stride - width) + stride + 1;
   viYUVSup.uDelta = ((stride - width) >> 1) + (stride >> 1) + 1;
   viYUVSup.vDelta = ((stride - width) >> 1) + (stride >> 1) + 1;
}
```

The **viYUVSetup** routine initializes the video parameters.

```
if( err = viYUVSetup(viInst,&viYUVSup) ) my_abort("viYUVSetup", err);
```

The **viStart** routine initializes image capture.

```
if (err = viStart(viInst))
   my_abort("viStart", err);
}
```

## Start Outputting an Image To Video Out (voYUVAPI )

The routine begins by initializing the video mode. The VO unit supports three output modes: cosited 4:2:2, interspersed 4:2:2, and 4:2:0 (see section 7-8 of the data book).

In cosited 4:2:2, the chrominance values (U and V) correspond to the first of two luminance values. In interspersed 4:2:2, they correspond to the midpoint between the two pixels. In 4:2:0 mode, there are four times fewer U and V than Y values (half as many as in 4:2:2). The chrominance values correspond to the point in the center of the square formed by consecutive horizontal and vertical pictures.

The VO unit has two interrupt modes. An interrupt can be generated at the end of the image area or when the scan line reaches a given value. The first is used.

```
void
voYUVAPI(voYUVModes_t mode,
   int imageWidth, int imageHeight, int imageStride,
   int imageVertOffset, int imageHorzOffset,
   Pointer yBase, Pointer uBase, Pointer vBase){
   tmLibdevErr_t err;

   memset((char *) (&voYUVSup), 0, sizeof (voYUVSetup_t));
   voYUVSup.mode = mode;
   voYUVSup.buf1emptyEnable = True;
   voYUVSup.yThresholdEnable = False;
   voYUVSup.yThreshold = False;
```

The next three lines set the Y, U, and V image pointers.

```
voYUVSup.yBase = yBase;
voYUVSup.uBase = uBase;
voYUVSup.vBase = vBase;
```

The **imageVertOffset** and **imageHorzOffset** correspond to the offset of the image from the top left hand corner of the active video area (see figure 7-12 of the data book).

```
voYUVSup.imageVertOffset = imageVertOffset;
voYUVSup.imageHorzOffset = imageHorzOffset;
```

The image height needs to be divided by two for interlaced scan. Lines of one field are interspersed with lines of another, so the stride needs to be doubled. The stride for U and V is half that for the Y pixels.

```
voYUVSup.imageHeight = (imageHeight >> 1);
voYUVSup.yStride     = (2 * imageStride);
voYUVSup.uStride     = imageStride;
voYUVSup.vStride     = imageStride;
```

The mode supplied to **voYUVSetup** is a combination of the VO mode and the use of 2x horizontal upscaling. The width of the image is halved in the presence of scaling.

```
switch( mode ){
   case vo422_COSITED_UNSCALED:
   case vo422_INTERSPERSED_UNSCALED:
   case vo420_UNSCALED:
      voYUVSup.imageWidth = imageWidth;
      break;
   case vo422_COSITED_SCALED:
   case vo422_INTERSPERSED_SCALED:
   case vo420_SCALED:
   default:
      voYUVSup.imageWidth = imageWidth << 1;
      break;
}
```

The call to **voYUVSetup** programs the 7185 registers.

```
 if( err = voYUVSetup(voInst,&voYUVSup) ) my_abort("voYUVSetup", err);
```

The call to **voStart** begins image display.

```
if( err = voStart(voInst) ) my_abort("voStart", err);
```

## Initialize Overlay Mode (voOverlayAPI )

**voOverlayAPI** copies the arguments into a structure and calls **voOverlaySetup**.

The TriMedia VO unit allows the display buffer to be overlaid with a raster in memory using alpha blending. Since the TriMedia processor uses a raster overlay the user has full control over the contents. For example, the overlay can contain a graphic logo as well as characters for teletext. The dimensions and position of the overlay with respect to the active image area are programmable.

The degree of blending is determined by the top three bits of two eight bit registers (GLOBAL ALPHA 0, GLOBAL ALPHA 1), as indicated in Table 7-4 of the data book. The TriMedia display buffer has separate Y, U, and V planes but the overlay raster is interspersed. Overlay images are stored in YVYU format. Figure 7-20 of the data book shows the format. The U and V values are the same for the two Y pixels.

The low order bit of U determines the alpha value for (Y0, U, V) (ALPHA 1, ALPHA 0) The low order bit of V determines the alpha value for (Y1, U, V) similarly.

The arguments to voOverlayAPI are the offset of the overlay from the left hand corner (**sLine**, **sPixel**), the size of the overlay (width, height), the values for alpha blending (alpha0, alpha1).

Because the overlaid image data is interspersed there is a single buffer pointer and stride (base and offset).

```
voOverlayAPI(int sLine, int sPixel, int width, int height,
   UInt alpha0, UInt alpha1, int offset, Pointer base){
   tmLibdevErr_t err;

   memset((char *) (&voOverlaySup), 0, sizeof (voOverlaySetup_t));
   voOverlaySup.overlayEnable = True;
   voOverlaySup.overlayStartY = sLine;
   voOverlaySup.overlayStartX = sPixel;
   voOverlaySup.overlayWidth = width;
   voOverlaySup.overlayHeight = height;
   voOverlaySup.alpha0 = alpha0;
   voOverlaySup.alpha1 = alpha1;
   voOverlaySup.overlayStride = offset;
   voOverlaySup.overlayBase = base;

   if err = voOverlaySetup(voInst,&voOverlaySup) )
      my_abort("voOverlaySetup", err);
}
```

## Inputting an Image for Display on VO (readYUVfiles)

There are no VO alignment constraints in image mode.

**vivoAlloc** calls **readYUVfiles** to read a 720 x 480 image in "tmlogo" into **bkbuf**.

```
err = readYUVFiles("tmlogo", 720, 480,
bkBuf[0].Y, bkBuf[0].U, bkBuf[0].V);

readYUVFiles(char *baseName, int hsize, int vsize,
             UInt32 ybuf, UInt32 ubuf, UInt32 vbuf){
   int             count, ySize, uvSize, row;
   char            fn[80];
   unsigned char  *pb;
   FILE           *fp;
```

The size of the UV buffer is half that of the Y buffer after conversion.

```
ySize = hsize * vsize;
uvSize = (ySize >> 1);
```

The Y data is in "**tmlogo.y**" The file is opened in binary mode.

```
sprintf(fn, "%s.y", baseName);
fp = fopen(fn, "rb");
if( !fp ) return (4);
```

The data is read into the buffer. For VO lines do not need to be aligned on cache line boundaries.

```
count = fread( (char*)ybuf, 1, ySize, fp );
fclose( fp );
```

The data is flushed back to the cache.

```
_cache_copyback(ybuf, ySize);
```

The U data is read from "**tmlogo.u**" in a similar fashion.

```
sprintf(fn, "%s.u", baseName);
fp = fopen(fn, "rb");
if( !fp ) return (4);
pb = (unsigned char *) ubuf;
count = 0;
```

There are half as many lines on the file as for the Y data. This is because the data is in 4:2:0 format.

```
for( row = 0; row < (vsize >> 1); row++ ){
```

There are half as many pixels per line as for the Y data also.

```
   count += fread(pb, 1, (hsize >> 1), fp);
```

The data from the odd field is reproduced for the even field also. The pointer is incremented to point to the next image. The data is flushed back for the cache.

```
   memcpy(pb + (hsize >> 1), pb, (hsize >> 1));
   pb += hsize;
}
_cache_copyback(ubuf, uvSize);
fclose(fp);
```

The code for reading the V data is the same as for the U data. The function returns zero to indicate successful completion.

```
return (0);
```

## ICP Setup

A color conversion filter is used to convert the captured image to overlay format. The input image is in CIF with the strides corresponding. The output stride is double the input stride because of 2x upscaling.

ICP setup requires opening an ICP instance (**icpOpen**) and associating an interrupt with it (**icpInstanceSetup**).

```
static void
SetupICP(){
   tmLibdevErr_t err;

   if (err = icpOpen(&icpInst)) my_abort("icpOpen", err);
   memset((char *) &icpInstSup, 0, sizeof (icpInstanceSetup_t));
   icpInstSup.interruptPriority = intPRIO_4;
   icpInstSup.isr = NULL;
   if( err = icpInstanceSetup(icpInst,&icpInstSup))
      my_abort("icpInstanceSetup", err);
```

The stride for Y is twice that for U and V since the image is in 4:2:2 format. The output stride is double that of the input since we are upscaling horizontally from 352 to 704.

```
memset( (char*)&icpImage, 0, sizeof(icpImageColorConversion_t) );
icpImage.yInputStride  = cifStride;
icpImage.uvInputStride = (cifStride >> 1);
icpImage.inputHeight   = cifHeight;
icpImage.inputWidth    = cifWidth;
icpImage.outputStride  = cifWidth<<1;
icpImage.outputHeight  = cifHeight;
icpImage.outputWidth   = cifWidth;
```

The **filterBypass** field can be **icpFILTER** or **icpBYPASS**. Bypass mode corresponds to simply picking the nearest pixel in the input for the output.

```
icpImage.filterBypass = icpFILTER;
```

The output is interspersed and the input is planar. The byte ordering is little endian on a Windows host, otherwise it is big endian. Output is to the SDRAM.

```
icpImage.outputPixelOffset = 0;
icpImage.inFormat = vdfYUV422Planar;
icpImage.outputDestination = icpSDRAM;
#ifdef __TCS_Win95__
   icpImage.littleEndian = True;
#else
   icpImage.littleEndian = False;
#endif
icpImage.outFormat = vdfYUV422Sequence;
```

## Buffer Processing for Full Resolution and CIF

Buffer processing is performed at the main level. The **mmBufUpdate** routine is called to process a captured image. the routine **mmBufUpdate** is used. It is called in the main level busy wait loop. **mmBufUpdate** first checks for a buffer ready.

```
void
mmBufUpdate(){
   int mmtmpNum;

   mmtmpNum = (mmNum + 1) % VID_NUMBUFS;
// If the buffer is ready
   if( genBuf[mmtmpNum].flag == VID_RDY_MM ){
```

The output buffer is made available for VO and the pointer is advanced to the next buffer. it is made available for VO.

```
genBuf[mmNum].flag = VID_RDY_VO;
mmNum = mmtmpNum;
```

## Buffer Processing for Overlay (mmOvlyBufUpdate)

**mmOvlyBufUpdate** first checks for a buffer.

```
void
mmOvlyBufUpdate(){
   int         mmtmpNum;
   tmLibdevErr_t err;
```

The arguments to the color conversion filter are the Y U and V pointers of the input buffer.

```
mmtmpNum = (mmNum + 1) % VID_NUMBUFS;
if (genBuf[mmtmpNum].flag == VID_RDY_MM)  {
```

The input image in buffer **mmNum**+1 is in planar format.

```
icpImage.yBase = (Pointer)genBuf[mmtmpNum].Y;
icpImage.uBase = (Pointer)genBuf[mmtmpNum].U;
icpImage.vBase = (Pointer)genBuf[mmtmpNum].V;
```

The output image in buffer **mmNum** is in interspersed format. A color conversion filter is used to converts. A busy wait loop is used to check for termination.

```
icpImage.outputImage = (Pointer)genBuf[mmNum].Y;
if (err = icpColorConversion(icpInst, &icpImage))
my_abort("icpColorConversion", err);
while( icpCheckBUSY() ){}
```

The output buffer is made available for VO and the pointer is advanced to the next buffer.

```
genBuf[mmNum].flag = VID_RDY_VO;
mmNum = mmtmpNum;
```

## VI Interrupt Service Routine (viTestISR)

The code has been edited to remove work-arounds for bugs for clarity. The function of **viTestISR** is to position a captured frame in the circular queue as ready for processing (VID_RDY_MM) as long as there is a buffer available for capture.

```
void
viTestISR(){
   unsigned long vi_status = MMIO(VI_STATUS);
   int           oddField;
   int           vitmpNum;
```

The VI interrupt service routine is a non interruptible handler.

```
#pragma TCS_handler
```

We determine whether the field is an even or an odd field.

```
oddField = viExtractODD(vi_status);
```

Potential interrupt sources include capture complete, under run, and highway bandwidth error. If this is a highway bandwidth error, we return without doing anything.

```
if (viHBE(vi_status)) {
   viAckHBE_ACK();
   return;
}
```

**capField** corresponds to CIF capture and overlay. If **capField** is non zero the even field is eliminated, effectively dividing by two the vertical resolution. The buffer pointer is advanced on reception of the odd field as long as there is an available buffer.

```
if( capField ){
   vitmpNum = (viNum + 1) % VID_NUMBUFS;
   if (oddField & (genBuf[vitmpNum].flag == VID_RDY_VI)) {
      genBuf[viNum].flag = VID_RDY_MM;
      viNum = vitmpNum;
      viYUVChangeBuffer(viInst,
      genBuf[viNum].Y,
      genBuf[viNum].U,
      genBuf[viNum].V);
   }
```

The captured buffer is acknowledged terminating interrupt processing.

```
   viAckCAP_ACK();
   return;
}
```

The rest of the routine corresponds to **capField** being zero. The code depends on whether we are processing the first (odd) or second (even) field of a frame. The following code corresponds to the case of an odd field.

```
if (firstField) {
```

The field flag is toggled.

```
firstField = False;
```

The if corresponds to a dropped field, which is an exception. This is the case if **firstField** and **oddField** differ. If this is so, the field is dropped, synchronizing the VI unit and the software.

```
if( !oddField ){
   /* skip even field to get sync */
   firstField = True;
} else {
```

The buffer pointers for the odd and even fields have a separation of one scan line.

```
   /* always start with odd field */
   viYUVChangeBuffer(viInst,
   genBuf[viNum].Y + yScanWidth,
   genBuf[viNum].U + uvScanWidth,
    genBuf[viNum].V + uvScanWidth);
}
```

The else case corresponds to the case of an even field. The buffer pointer is advanced if there is an available buffer.

```
}else{
   vitmpNum = (viNum + 1) % VID_NUMBUFS;
   if (genBuf[vitmpNum].flag == VID_RDY_VI) {
      genBuf[viNum].flag = VID_RDY_MM;
      viNum = vitmpNum;
   }
```

The buffer pointers are reset to the beginning of the buffer.

```
viYUVChangeBuffer(viInst,
genBuf[viNum].Y,
genBuf[viNum].U,
genBuf[viNum].V);
```

The field flag is toggled.

```
firstField = 1;
```

The capture is acknowledged ending interrupt processing.The video out interrupt source routine is similar to the one explained in the ICP example.

```
viAckCAP_ACK();
```

The video out interrupt service routine is similar to the one explained in the ICP example.

## Querying the Configuration

**Reportsys** calls the HAL functionality **procGetCapabilities** to identify the processor type. **procGetCapabilities(&procCap)**; The following structure is returned.

The fields of the data structure identify the processor type (TM1000, TM1100, etc.), the processor version, the revision ID, and the clock frequency in hertz.

By TriMedia API convention, there are two types that are defined (for the structure and for a pointer); the version number is the first word of each structure. The last three fields identify the type of host and the processor configuration, for a multiprocessor.

```
typedef struct{
   tmVersion_t    version;           /* version of this sw module */

   procDevice_t   deviceID;          /* for implemented functionality */
   procRevision_t revisionID;        /* for bugs, performance, etc. */
   UInt32         cpuClockFrequency; /* in Hz */
   UInt32         nodeNumber;        /
* node number in case of multiple TMs */
   UInt32         numberOfNodes;     /* number of TMs in system */
   tmHostType_t   hostID;            /* tmInvalidHost, tmNoHost, tmTmSimHost,
                                      * tmWin32Host, or tmMacOSHost */
} procCapabilities_t, *pprocCapabilities_t;
```

This terminates the vivot example. Examples of how to use VI and VO in raw and message-passing modes are available in the Power on Self Test (POST). The following chapter contains more information on how to use the video units with the ICP and VGA cards.

Chapter 5

# Programming TriMedia Video Applications Using the ICP TSSA API

## Introduction

This chapter describes how to write video applications using the ICP-based Video Transformer. For a detailed description of this API, see Chapter 5, *Image Co-Processor (ICP) API*, of Book 7, *Video Support Libraries*.

The Video Transformer is designed to simplify the use of the Image Co-Processor (ICP) peripheral. This component offers a number of advantages over the tmICP device library. Several tasks may each open an instance of the Video Transformer and issue requests for video filtering; the component library will queue up the requests and issue them one by one to the ICP. The required vertical, horizontal, and color conversion filter operations to perform a transformation are automatically calculated and issued to the ICP. All buffers required to store scaled intermediate images are created and destroyed automatically. The component also supports antiflicker filtering for DSPCPU generated graphics and deinterlacing for interlaced to progressive scan conversion.

The AL layer supports non-data streaming (*push* mode), while the OL layer supports data streaming (*pull* mode).

## The exolVtransICP Example Program

The exolVtransICP example demonstrates the use of the OL layer of the Video Transformer. The example simply connects an instance of the Video Digitizer to an instance of the Video Transformer. The digitizer captures live data using the video-in device while the transformer scales the image, converts it from YUV to RGB, and then displays it on the PC screen via the PCI interface. The user may specify parameters on the console to enable antiflicker filtering and deinterlacing.

The source code for this example is contained within the examples/exolVtransICP directory of the application tree. The example will now be described, with emphasis placed on the Video Transformer aspects. We recommend that you first read Chapter 4, *Programming TriMedia Video Applications* as it describes the use of the Video Digitizer. Chapter 19, *TMBoard API* of Book 5, *System Utilities* provides additional information on this example and a separate AL layer example (examples/exalVtransICP).

### Include Files

```
#include <tm1/tmAvFormats.h>
#include "tmos.h"
#include "tmolVtransICP.h"
#include "tmolVdigVI.h"

#include <stdio.h>
#include <tmlib/dprintf.h>      /* for debugging with DP(()) */

#include "sys_conf.h"
```

The tmAvFormats.h file contains the definitions for the packets which are used to store video data. The type definitions and function prototypes for the Video Transformer are defined in tmolVtransICP.h.

## Definitions

```
#define VIDEO_ADDR 0xe0000000 /* Default Start address of the screen    */
#define VIDEO_STRIDE 2048     /* For 24 bit video it is 3x screen width */
#define VIDEO_MODE   3        /* RGB15+Alpha                            */

/* video in image format */
#define  INPUT_HEIGHT   480
#define  INPUT_WIDTH    720
#define  INPUT_STRIDE   768

/* * Video out image format */
#define  OUTPUT_HEIGHT   360
#define  OUTPUT_WIDTH    540
#define  OUTPUT_STRIDE   VIDEO_STRIDE
```

The default address of the PCI video card, the stride of the video card, and the RGB format are defined. Note that these are simply the default parameters and the user must specify the correct parameters via the command line.

The height, width, and stride of the captured image are defined using the **INPUT_HEIGHT**, **INPUT_WIDTH**, and **INPUT_STRIDE** respectively.

The **OUTPUT_WIDTH** and **OUTPUT_HEIGHT** specify the size of the image which will be displayed on the PC screen. The **OUTPUT_STRIDE** will be equal to the stride of the PCI video card.

## Static Variables

```
/* These command line args come from the modified sysinit.c which allows the
 * task to read the required parameters. */
   extern int __argc;
   extern char **__argv;
```

The **__argc** and **__argv** variables are used to pass command line arguments to the application. These arguments will consist of the PCI video card address, the display stride, and the display RGB format.

## Specifying the Packet Format

```
static tmVideoFormat_t digitizerFormat = {
   sizeof(tmVideoFormat_t), /* size           */
   0,                       /* hash           */
   0,                       /* referenceCount */
   avdcVideo,               /* dataClass      */
   vtfYUV,                  /* dataType       */
   vdfYUV422Planar,         /* dataSubtype    */
   vdfInterlaced,           /* description    */
   INPUT_WIDTH,             /* imageWidth;    */
   INPUT_HEIGHT,            /* imageHeight;   */
   INPUT_STRIDE,            /* imageStride;   */
};
```

This structure defines the format of the packets used to transfer data between the Video Digitizer and Video Transformer. The **hash** and **referenceCount** fields are used exclusively by the format manager, and must be set to zero.

The **dataClass** and **dataSubtype** must be set to **avdcVideo** and **vtfYUV** respectively. The **dataSubtype** may be set to either **vdfYUV422Planer** or **vdfYUV420Planer**. For this example, YUV422 video is used.

The **description** field specifies that the data stored in the packet buffers is interlaced. The even and odd fields are stored in the same buffer using an interleaved format.

Finally, the captured frame height, width, and stride are defined.

## Specifying the Output Format

```
static tmVideoFormat_t outputFormat = {
   sizeof(tmVideoFormat_t), /* size           */
   0,                       /* hash           */
   0,                       /* referenceCount */
   avdcVideo,               /* dataClass      */
   vtfRGB,                  /* dataType       */
   vdfRGB15Alpha,           /* dataSubtype    */
   0,                       /* description    */
   OUTPUT_WIDTH,            /* imageWidth;    */
   OUTPUT_HEIGHT,           /* imageHeight;   */
   OUTPUT_STRIDE,           /* imageStride;   */
};
```

The **outputFormat** structure specifies the format of the Video Transformer output. The component is capable of writing its output to either SDRAM or PCI. For output to SDRAM, the processed data will be placed in a packet. For output to PCI, the data will be stored in the PCI video card memory and no packet will be used. In either case, the output format must be specified using a **tmVideoFormat_t** structure.

The **dataClass** field must always be set to **avdcVideo**. The **dataType** field may be either **vtfYUV** or **vtfRGB** when the output is to SDRAM. When writing to PCI, the output must be **vtfRGB**.

The **dataSubtype** depends upon the **dataType** field. For YUV data it can be **vdfYUV422Planer**, **vdfYUV420Planer**, **vdfYUV422Sequence**, or **vdfYUV422SequenceAlpha**. For RGB, it can be **vdfRGB8A_233**, **vdfRGB8R_332**, **vdfRGB15Alpha**, **vdfRGB16**, **vdfRGB24**, or **vdfRGB24Alpha**. As the Video Transformer will be writing to the PC screen, the output must be RGB. The subtype will be specified by the user via the command arguments.

The **description** field is set to zero as the component does not use this value on its output.

Finally, the output height, width, and stride are specified.

## Packet Defines and Function Prototypes

```
#define NUMPACKETS  4
#define NUMBUFFERS  3   /* Y, U, V */

/* function prototypes */

extern void
get_parameters(Int argc, Char * argv[],
Int * disp_addr, Int * stride, Int * mode);

extern tmLibappErr_t
tmalVtransICPProgress(Int instId, UInt32 flags, ptsaProgressArgs_t args);

extern tmLibappErr_t
tmalVtransICPCompletion(Int instId, UInt32 flags, ptsaCompletionArgs_t args);
```

The example uses four packets (**NUMPACKETS**) to exchange video frames between the digitizer and transformer. Each packet contains three buffers (**NUMBUFFERS**) which store the Y, U, and V data.

The **get_parameters()** function is used to obtain the user-specified command line arguments. This function will not be described.

The **tmalVtransICPProgress()** and **tmalVtransICPCompletion()** callback functions will be used by the Video Transformer to report information to the application. These will be described later.

## Variables

```
void tmosMain(){
   tmLibappErr_t  rval;
   Int            digitizerInstance;

   ptmolVdigVIInstanceSetup_t    digitizerInstSetup;
   Int                           vtrans0Instance;
   ptmolVtransICPInstanceSetup_t vtransInstSetup;
   ptsaControlDescriptor_t       vtransCommand;
   tsaControlDescriptorSetup_t   csetup;

   ptsaInOutDescriptor_t         iodesc;
   ptsaInOutDescriptorSetup_t    ioSetup;

   ptmolVdigVICapabilities_t    digitizerCap;
   ptmolVtransICPCapabilities_t vtransCap;

   char   ins[80];
   Int    pciAddress;
   Int    pciStride;
   Int    videoMode;

   tsaControlArgs_t  controlArgs;
   Bool              quitDetected = False;
   Bool              antiflickerEnable = False;
   Bool              deinterlaceEnable = False;
```

The **rval** variable is used to store the values returned by calls to the Video Digitizer, Video Transformer, and tsaDefaults library. Error values are defined in tmLibappErr.h, with a return value of **TMLIBAPP_OK** indicating no error.

The **digitizerInstance** variable is used to store the instance id of the Video Digitizer, while **digitizerInstSetup** is a pointer to the setup structure which will be used to configure the digitizer.

The **vtrans0Instance** variable will be used to store the instance id of the Video Transformer. The component enables up to four instances to be open. The **vtransInstSetup** variable points to the component's setup structure and will be used to configure the instance. The **vtransCommand** variable is a pointer to a control descriptor. The Video Transformer allows the application to send configuration commands to it while it is streaming data. The control descriptor is used to specify the message interface between the application and the instance of the transformer. The **csetup** structure specifies parameters that are used when the control descriptor is created.

The connection between the Video Digitizer and Video Transformer is specified using a **tsaInOutDescriptor**. This describes the connection and the packets that will be used to transfer data.

The capabilities of the two components will be pointed to using **digitizerCap** and **vtransCap**. These will be used by the format manager to ensure that the two components can communicate with each other.

The **ins[80]** char array is used to store character commands entered by the user.

The **pciAddress**, **pciStride**, and **videoMode** are used to store information concerning the PCI video card. These will be initialized via the command arguments.

The **controlArgs** structure is used to pass control information from the application to the component instance. This will be described in more detail in the section “User Input” beginning on page -54.

Finally, the **quitDetected**, **antiflickerEnable**, **deinterlaceEnable** are boolean flags. The **quit-Detected** flag is used to indicate that the user has typed an exit command. The **antiflick-erEnable** and **deinterlaceEnable** are flags that indicate whether the antiflicker filter and deinterlace filter are enabled.

## Initialization

```
DPmode(DP_PERSIST);
DPsize(1024*1024);

tmosInit();

printf("TriMedia OS Video Transformer Demo. v1.0\n");
printf("\nThis program uses the video digitizer and video transformer\n");
printf("to pass video in to the PCI video.\n");
printf("The program is compiled to support NTSC and CVBS.\n");
printf("Recompile to change this.\n\n");

/* get parameters from the command line */

get_parameters(__argc, __argv, &pciAddress, &pciStride, &videoMode);

if (videoMode == 1)
   outputFormat.dataSubtype = vdfRGB24Alpha;
else if (videoMode == 2)
   outputFormat.dataSubtype = vdfRGB24;
else if (videoMode == 3)
   outputFormat.dataSubtype = vdfRGB15Alpha;
else if (videoMode == 4)
   outputFormat.dataSubtype = vdfRGB16;

outputFormat.imageStride = pciStride;
```

The **DPmode** and **DPsize** functions are used to specify the debug print buffer. This buffer facilitates debugging and stores information that is written to it by either the application or the component instances.

The **tmosInit** function will initialize the multi-tasking operating system. In this example, the application executes in the default task, while a separate task will be created automatically for the Video Transformer instance. The Video Digitizer is an interrupt-based component and, therefore, does not have a separate task.

The command line parameters are read from arguments passed down to the example program. The user must specify the PCI video address, the PCI stride, and the PCI screen mode. The user will enter the screen mode as a value from one to four and this is re-mapped to the corresponding **tmAvFormat_t** type.

## Get Capabilities

```
printf("Getting VdigVI Capabilities\n");
if(rval = tmolVdigVIGetCapabilities(&digitizerCap)) {
   printf("Error in tmolVdigVIGetCapabilities: 0x%x\n",rval);
   tmosExit(-1);
}
printf("Getting VtransICP Capabilities\n");
if(rval = tmolVtransICPGetCapabilities(&vtransCap)) {
   printf("Error in tmolVtransICPGetCapabilities: 0x%x\n",rval);
   tmosExit(-1);
}
```

The capabilities of the two components must be obtained before the **tsaInOutDescriptor** is created. This information will be used to ensure that they are compatible.

## Make the Connection Between the Two Components

```
ioSetup=(ptsaInOutDescriptorSetup_t)malloc(sizeof(tsaInOutDescriptorSetup_t)
       + (NUMBUFFERS-1)*sizeof(UInt32));
ioSetup->format          = (ptmAvFormat_t)(&digitizerFormat);
ioSetup->flags           = tsaIODescSetupFlagCacheMalloc;
ioSetup->fullQName       = "VDF0";
ioSetup->emptyQName      = "VDE0";
ioSetup->queueFlags      = tmosQueueFlagsStandard;
ioSetup->senderCap       = digitizerCap->defaultCapabilities;
ioSetup->receiverCap     = vtransCap->defaultCapabilities;
ioSetup->senderIndex     = VDIGVI_MAIN_OUTPUT;
ioSetup->receiverIndex   = VTRANSICP_MAIN_INPUT;
ioSetup->packetBase      = 0x100;
ioSetup->numberOfPackets = NUMPACKETS;
ioSetup->numberOfBuffers = NUMBUFFERS;
ioSetup->bufSize[0]      = INPUT_HEIGHT * INPUT_STRIDE;      /* Y */
ioSetup->bufSize[1]      = INPUT_HEIGHT * INPUT_STRIDE / 2;  /* U */
ioSetup->bufSize[2]      = INPUT_HEIGHT * INPUT_STRIDE / 2;  /* V */

/* Create InOutDescriptor */
printf("Creating InOutDescriptor\n");
if(rval = tsaDefaultInOutDescriptorCreate(&iodesc, ioSetup)) {
   printf("Error in tsaDefaultInOutDescriptorCreate: 0x%x\n",rval);
   tmosExit(-1);
}
```

A **tsaInOutDescriptor** setup structure is created and initialized. This is similar to the connection setup described in the section *Make the Connection Between the Two Components* described in Chapter 4. The difference being that the Video Transformer capabilities are passed as the **receiverCap**. Note that the packets that will be placed in the empty queue will have an id beginning with **0x100**; i.e. the four packets will have the following id's: **0x100**, **0x101**, **0x102**, and **0x103**.

## Create the Video Transformer Control Descriptor

```
csetup.commandQName  = "vtOC";
csetup.responseQName = "vtOR";
csetup.queueFlags    = tmosQueueFlagsStandard;
csetup.flags         = tsaIODescSetupFlagNone;
if(rval = tsaDefaultControlDescriptorCreate(&vtransCommand, &csetup)) {
   tmAssert((rval == TMLIBAPP_OK), rval);
}
```

The application may send configuration commands to the Video Transformer using a control descriptor. A setup structure must first be initialized before the descriptor is created. The **commandQName** and **responseQName** fields specify a four letter name which will be associated with the command and response queues; this may be used for debugging purposes. The **queueFlags** specify information used for message queue creation. The **tmolQueueFlagsStandard** flags specify that the queues will be local to the processor, and there is no limit to the number of messages which can be placed on them. The **flags** field is currently unused and should be set to **tsaIODescSetupFlagNone**.

The **tsaDefaultControlDescriptorCreate** function will allocate memory for the control descriptor, initialize the relevant values, and create the message queues.

## Setup the Video Digitizer

```
/* setup video input digitizer */
rval = tmolVdigVIOpen(&digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVdigVIGetInstanceSetup(digitizerInstance,&digitizerInstSetup);
tmAssert((rval == TMLIBAPP_OK), rval);

digitizerInstSetup->instSetup->outputDescriptors[VDIGVI_MAIN_OUTPUT]
                                                               = iodesc;
rval = tmolVdigVIInstanceSetup(digitizerInstance, digitizerInstSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("digitizer initialized.\n");
```

An instance of the Video Digitizer is first opened. It is important that the application check the return value of this function. A typical error would be **TMLIBAPP_ERR_MODULE_IN_USE**, which indicates that another task has already opened an instance. The digitizer supports only a single instance.

The **tmolVdigVIGetInstanceSetup** function should be called to obtain a pointer to the instance setup structure. This will be used to configure the instance. The output descriptor is set to point to the **InOutDescriptor** created previously.

Finally, the **tmolVdigVIInstanceSetup** function is called to configure the instance.

## Setup the Video Transformer

```
rval = tmolVtransICPOpen(&vtrans0Instance);
tmAssert((rval == TMLIBAPP_OK), rval);

rval = tmolVtransICPGetInstanceSetup(vtrans0Instance,&vtransInstSetup);
tmAssert((rval == TMLIBAPP_OK), rval);

/* Queues have to be initialized. We are using only the main input, but no
 * overlay inputs. As we are using the PCI for output we have no output queue/
 * pin. By default, unused pins will are set to Null. */
vtransInstSetup->defaultSetup->inputDescriptors[VTRANSICP_MAIN_INPUT]
                                                                  = iodesc;
vtransInstSetup->defaultSetup->controlDescriptor = vtransCommand;
vtransInstSetup->defaultSetup->progressFunc      = tmalVtransICPProgress;
vtransInstSetup->defaultSetup->completionFunc    = tmalVtransICPCompletion;

/* setup the PCI output image parameters */
vtransInstSetup->outputFormat = outputFormat;

vtransInstSetup->outputDest    = tmalVtransICPPCI;
vtransInstSetup->outputPCIAddr = (UInt8 *) pciAddress;

vtransInstSetup->deinterlaceEnable = False;
vtransInstSetup->antiflickerEnable = False;

rval = tmolVtransICPInstanceSetup(vtrans0Instance, vtransInstSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("transformer instance 0 initialized.\n");
```

An instance of the Video Transformer is opened. Up to four instances may be open at any instant of time. The instance setup structure is obtained by calling **tmolVtransICPGetInstanceSetup** and this structure will be used to specify the initial configuration. The main image input descriptor is set to the **InOutDescriptor** that was created before. In the example, the overlay input is not used and, therefore, no setup information is specified. The **controlDescriptor** is initialized with the control descriptor.

The **progressFunc** and **completionFunc** callback functions are set to point to functions contained within the example program. The Video Transformer will call the application's progress function when an image transformation request has been placed on the ICP queue. The completion function will be called once the transformation request has been processed. These callback functions are optional.

The output parameters specify the output format and the destination of the video transformation. In this example, the output is to PCI, so it is necessary for the application to specify the **outputDestination** as **tmalVtransICPPCI**, and the **outputPCIAddr** to the address of the PCI video memory. It is also necessary to initialize the output format as this specifies the image output parameters.

If the output was to SDRAM, then an **InOutDescriptor** must be created which connects the output of the Video Transformer to the input of another component. The **outputDestination** should be set to **tmalVtransICPSDRAM**, with the **outputPCIAddr** and **outputFormat** set to Null. In this mode, the instance will obtain the output format from the output descriptor.

The **deinterlaceEnable** and **antiflickerEnable** flags are set to disabled for the initial configuration.

Finally, the **tmolVtransICPInstanceSetup** function is called to transfer the setup parameters to the instance.

## Starting the Component Instances

```
DP(("\nStarting Video transformer\n"));
rval = tmolVtransICPStart(vtrans0Instance);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("transformer started.\n");

DP(("\nStarting Video Digitizer\n"));
rval = tmolVdigVIStart(digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("digitizer started.\n");
```

Data streaming is initiated by calling the start function for the two instances. These functions are **tmolVdigVIStart** and **tmolVtransICPStart** respectively. The Video Digitizer executes entirely in an interrupt service routine, while the Video Transformer instance executes within its own task.

## User Input

```
printf("\nVideo transformer demo started.\n");
printf("Video input is being echoed to video output.\n");
printf("\nThe following commands are available:\n");
printf("\tA - toggle antiflicker filter\n");
printf("\tD - toggle deinterlace filter\n");
printf("\tI - disable both antiflicker and deinterlace filters\n");
printf("\tQ - quit\n");

printf("Press return after entering the required option \n");

while (!quitDetected) {
   gets(ins);
   switch( ins[0] ){
```

The user may enter commands via the console to control the operation of the Video Transformer. The ‘A’ key will toggle the antiflicker filter, the ‘D’ key will toggle the deinterlace filter, and the ‘I’ key will disable the antiflicker and deinterlace filters if they are enabled. The ‘Q’ key will cause the program to exit.

The input parsing uses a simple switch statement to interpret the commands.

```
   case ’a’:
   case ’A’:
/* disable the deinterlace if it is enabled */
   if (deinterlaceEnable) {
      deinterlaceEnable = False;
      controlArgs.command =  VTRANS_CONFIG_DEINTERLACE_ENABLE;
      controlArgs.parameter = (Pointer) &deinterlaceEnable;
      controlArgs.timeout = 0;
      rval = tmolVtransICPInstanceConfig( vtrans0Instance, tsaControlWait,
                                          &controlArgs);
      tmAssert(rval == TMLIBAPP_OK, rval);
      printf("Disabled Deinterlace filter\n");
   }

   antiflickerEnable ^= 1;
   controlArgs.command =  VTRANS_CONFIG_ANTIFLICKER_ENABLE;
   controlArgs.parameter = (Pointer) &antiflickerEnable;
   controlArgs.timeout = 0;
   rval = tmolVtransICPInstanceConfig( vtrans0Instance, tsaControlWait,
                                       &controlArgs);
   tmAssert(rval == TMLIBAPP_OK, rval);

   if( antiflickerEnable) {
      printf("Enabled antiflicker filter\n");
   }else{
      printf("Disabled antiflicker filter\n");
   }
   break;
```

When the antiflicker key is entered, a check is made to see if the deinterlace filter is enabled. If it is, the application will disable it. Note that this is not a restriction of the Video Transformer component, which is able to do both deinterlacing and antiflicker filtering—it is simply made to be mutually exclusive in the application. Deinterlacing will be disabled by setting up a **controlArgs** structure with the relevant command and command parameter. The command is **VTRANS_CONFIG_DEINTERLACE_ENABLE** in this case, and the parameter will be the **deinterlaceEnable** flag, which was set to false. The timeout field specifies the time the configuration function should wait before returning a timeout error. In this case, the value of zero indicates that the function should wait until it receives a response. It then calls the **tmolVtransICPInstanceConfig** function to perform the configuration.

The **antiflickerEnable** flag is toggled, and the control arguments structure initialized. The command field is set to **VTRANS_CONFIG_ANTIFLICKER_ENABLE** and a call is made to **tmolVtransICPInstanceConfig**.

```
case 'd':
case 'D':
/* Disable antiflicker if it is enabled */
   if (antiflickerEnable) {
      antiflickerEnable = False;
      controlArgs.command =  VTRANS_CONFIG_ANTIFLICKER_ENABLE;
      controlArgs.parameter = (Pointer) &antiflickerEnable;
      controlArgs.timeout = 0;
      rval = tmolVtransICPInstanceConfig( vtrans0Instance, tsaControlWait,
                                          &controlArgs);
      tmAssert(rval == TMLIBAPP_OK, rval);
      printf("Disabled antiflicker filter\n");
   }
   deinterlaceEnable ^= 1;
   controlArgs.command =  VTRANS_CONFIG_DEINTERLACE_ENABLE;
   controlArgs.parameter = (Pointer) &deinterlaceEnable;
   controlArgs.timeout = 0;
   rval = tmolVtransICPInstanceConfig( vtrans0Instance, tsaControlWait,
                                       &controlArgs);
   tmAssert(rval == TMLIBAPP_OK, rval);
```

```
    if (deinterlaceEnable) {
       printf("Enabled Deinterlace filter\n");
    }else{
       printf("Disabled Deinterlace filter\n");
    }
    break;
```

The deinterlace toggle operates in a similarly to the antiflicker toggle. If the antiflicker filter is enabled, it is switched off using the **tmolVtransICPInstanceConfig** function.

The deinterlace enable flag is then toggled and the **VTRANS_CONFIG_DEINTERLACE_ENABLE** command is sent to the transformer instance.

```
case 'i':
case 'I':
/* Disable antiflicker and deinterlace (ie. display interlaced) */
    antiflickerEnable = False;
    controlArgs.command =  VTRANS_CONFIG_ANTIFLICKER_ENABLE;
    controlArgs.parameter = (Pointer) &antiflickerEnable;
    controlArgs.timeout = 0;
    rval = tmolVtransICPInstanceConfig( vtrans0Instance, tsaControlWait,
                                        &controlArgs);
    tmAssert(rval == TMLIBAPP_OK, rval);
    printf("Disabled antiflicker filter\n");

    deinterlaceEnable = False;
    controlArgs.command =  VTRANS_CONFIG_DEINTERLACE_ENABLE;
    controlArgs.parameter = (Pointer) &deinterlaceEnable;
    controlArgs.timeout = 0;
    rval = tmolVtransICPInstanceConfig( vtrans0Instance, tsaControlWait,
                                        &controlArgs);
    tmAssert(rval == TMLIBAPP_OK, rval);
    printf("Disabled Deinterlace filter\n");
    break;
```

The 'interlace' command simply switches off both the antiflicker and deinterlace filters using two calls to the **tmolVtransICPInstanceConfig** function. In this mode, the Video Transformer instance will only perform scaling and color conversion on the captured video frames.

```
    case 'q':
    case 'Q':
       DP(("User requested to quit the example\n"));
       quitDetected = True;
       break;

    default:
       break;
    }
}
```

Once the user enters the quit command from the console, the **quitDetected** flag will be set, which causes the main **while** loop to be exited.

## Stop and Shutdown

```
printf("\nStopping video transformer instance 0\n");
DP(("\nStopping video transformer instance 0\n"));
rval = tmolVtransICPStop(vtrans0Instance);
tmAssert(rval == TMLIBAPP_OK, rval);

printf("\nStopping video digitiser\n");
DP(("\nStopping video digitiser\n"));
rval = tmolVdigVIStop(digitizerInstance);
tmAssert(rval == TMLIBAPP_OK, rval);

tmolVdigVIClose(digitizerInstance);
rval = tmolVtransICPClose(vtrans0Instance);
tmAssert(rval == TMLIBAPP_OK, rval);

/* Check we have the correct number of packets left in the queues */
rval = tsaDefaultCheckQueues(iodesc);
printf("tsadefaultCheckQueues() returned 0x%x\n", rval);

/* Destroy InOutDescriptors and command queues */
printf("Destroying InOutDescriptors\n");
if(rval = tsaDefaultInOutDescriptorDestroy(iodesc)) {
printf("Error in tsaDefaultInOutDescriptorDestroy: 0x%x\n",rval);
tmosExit(-1);
}

rval = tsaDefaultControlDescriptorDestroy(vtransCommand);
tmAssert(rval == 0, rval);

DP(("Demo Complete.\n"));
printf("Demo Complete.  \n");
tmosExit(0);
```

Data streaming is terminated by calling the stop functions of each component. When **tmolVtransICPStop** is called, the Video Transformer instance will return any packets it may have in its possession, it then calls the completion function, and suspends its task. When **tmolVdigVIStop** is called, it will stop video capture and return the packet that it had in its possession.

The two instances are then closed by calling **tmolVdigVIClose** and **tmolVtransICPClose** respectively. Closing the Video Transformer will destroy the transformer instances task.

It is recommend that the application call the **tsaDefaultCheckQueues** function to ensure that the correct number of packets have been left in the **InOutDescriptor**.

Calling **tsaDefaultInOutDescriptorDestroy** will remove all packets from the descriptor queues, free up their data buffers, and free the space allocated for the descriptor.

Finally, the Video Transformer control descriptor should be destroyed by calling the **tsa-DefaultControlDescriptorDestory** function.

## Application Progress Function

```
tmLibappErr_t
tmalVtransICPProgress(Int instId, UInt32 flags, ptsaProgressArgs_t args){
   DP(("tmalVtransICPProgress[%x]: inside callback!\n", instId));
   return (TMLIBAPP_OK);
}
```

The application may supply a progress function to the Video Transformer instance. This function will be called by the Video Transformer once a packet has been placed on the ICP request queue. The example progress function simply prints a message to the DP debug buffer.

## Application Completion Function

```
tmLibappErr_t
tmalVtransICPCompletion(Int instId, UInt32 flags, ptsaCompletionArgs_t args){
   DP(("tmalVtransICPCompletion[%x]: inside callback!\n", instId));
   return (TMLIBAPP_OK);
}
```

The application may supply a completion function to the transformer instance. This function will be called once a frame has been processed by the ICP. It will also be called after the instance has been asked to stop. The example completion function prints a message to the DP debug buffer.

Chapter 6

# Programming TriMedia Audio Applications

## Introduction

This chapter describes how to write an audio application using the range of programming interfaces available on TriMedia. For a detailed description of these APIs, refer to Book 6, *Audio Support Libraries*, especially the sections on the audio renderer, audio digitizer, and the audio device library.

This chapter begins by describing a high level interface to the audio system. The audio renderer and the audio digitizer modules provide a high level interface to TriMedia audio services. These are fully compatible with other useful libraries, such as the Dolby AC3 and ProLogic decoders, and the audio mixers.

Next the reader is introduced to the audio device libraries that underlie the renderer and digitizer. Finally, the foundation provided by the board support library is briefly discussed.

## TSSA Audio Modules

TriMedia software modules are constructed to a specification known as the TriMedia Streaming Software Architecture (TSSA). This software architecture is documented in *Book 3, Software Architecture*. While the present chapter is easily intelligible without a background in TSSA, users will find it helpful to read about TSSA before starting serious programming.

The audio system on TriMedia is built in layers. Since the highest layer has the most functionality, this discussion will start at the top and work its way down.

A number of audio modules are available for use on TriMedia. These include the audio renderer and audio digitizer, which are used for audio playback and capture, respectively. A Dolby AC3 decoder and a Dolby ProLogic decoder are available. An example of a simple audio mixer is provided with source code. And the DTV demonstration application includes an audio system that connects all of these together. In addition, MPEG audio decoders and G.723 audio codecs are available as portions of the DVD player and the Video Phone packages, respectively. The DTV demonstration is constructed using TSSA-compatible libraries. The DVD and Video Phone libraries are not yet TSSA-compliant.

### The Audio Renderer

This chapter is an overview of the audio renderer. A detailed reference to the API of the audio renderer is provided in Chapter 5, “Audio Renderer (ArendAO) API,” of *Software Library APIs*.

The audio renderer is designed to make it easy to play audio on TriMedia. The audio renderer installs an interrupt service routine and uses it to play buffers of audio. The audio renderer is a high level interface that is uniform across different hardware implementations.

The audio renderer can, in fact, be run in two different modes. These are sometimes known as *push* mode and *pull* mode. In the push mode, no operating system dependencies exist, and a simple function is used to copy audio to the output. This is the push, from application to renderer. While this model is easy to understand, it does not lend itself to expansion. In particular, many details of operation are left to the application. A higher level interface standardizes many of the details of data exchange in order to eliminate the duplication of code. A demonstration of the push model is available in the exalArendAO demonstration program.

When the pull model is used to render audio, the producer of audio places buffers full of data into a queue. Empty packets are available in another queue. Since the application is driven by the need for empty packets, we say that it “pulls” packets from the empty queue. Several demonstration programs illustrate the use of the audio renderer in this mode. We will first discuss the one known as exolArendAO.

It is easiest to demonstrate the audio renderer by connecting it to a file reader. This lets you play audio files. An illustration of the code used for this task is shown in its entirety below. Directly following, each part of code is examined and discussed.

```
void ARendFilePlay(char *fileName){
   tmLibappErr_t                err;
   Int                          readerInstance, arendInstance;
   Char                         ins[80];
   ptmolArendAOInstanceSetup_t arSetup;
   ptmolFreadInstanceSetup_t   frSetup;
   ptmAudioFormat_t             paf;
   tmAudioFormat_t              audioFormat;
   ptmolFreadCapabilities_t    frCaps;
   ptmolArendAOCapabilities_t  arCaps;
   tsaInOutDescriptorSetup_t   iodSetup;
   ptsaInOutDescriptor_t        iod;

 /* find out what formats are supported */
   tmolFreadGetCapabilities(&frCaps);
   tmolArendAOGetCapabilities(&arCaps);
    if (!(paf->dataSubtype & apfStereo16)) {
      printf("Stereo audio playback not supported on this board.\n");
      return;
   }

/* Open the components involved and get their setup structures */
   err = tmolFreadOpen(&readerInstance);
   tmAssert((err == TMLIBAPP_OK), err);
   err = tmolFreadGetInstanceSetup(readerInstance, &frSetup );
   tmAssert((err == TMLIBAPP_OK), err);

   err = tmolArendAOOpen(&arendInstance);
   tmAssert((err == TMLIBAPP_OK), err);
   err = tmolArendAOGetInstanceSetup(arendInstance, &arSetup );
   tmAssert((err == TMLIBAPP_OK), err);

/* describe the connection between the two components */
/* assemble audio format */
   audioFormat.size         = sizeof(tmAudioFormat_t);
   audioFormat.hash         = audioFormat.referenceCount = 0;
   audioFormat.dataClass    = avdcAudio;
   audioFormat.dataType     = atfLinearPCM;
   audioFormat.dataSubtype  = apfStereo16;
   audioFormat.description  = 16;
   audioFormat.sampleRate   = sRate;
/* create an InOutDescriptor */
   iodSetup.format          = (ptmAvFormat_t)&audioFormat;
   iodSetup.flags           = tsaIODescSetupFlagCacheMalloc;
   iodSetup.fullQName       = "full";
   iodSetup.emptyQName      = "mpty";
   iodSetup.queueFlags      = tmosQueueFlagsStandard;
   iodSetup.senderCap       = frCaps->defaultCapabilities;
   iodSetup.receiverCap     = arCaps->defaultCapabilities;
   iodSetup.senderIndex     = 0;
   iodSetup.receiverIndex   = 0;
   iodSetup.packetBase      = 0;
   iodSetup.numberOfPackets = NUMBER_OF_PACKETS;
   iodSetup.numberOfBuffers = 1;
   iodSetup.bufSize[0]      = 2 * sizeof(Int16) * BUFSIZE;
   err = tsaDefaultInOutDescriptorCreate(&iod, &iodSetup);
   tmAssert((err == TMLIBAPP_OK), err);
 /* setup file reader */
   frSetup->defaultSetup->outputDescriptors[0] = iod;
   frSetup->defaultSetup->priority = READER_PRIORITY;
   frSetup->fileName               = fileName;
   printf("Opening %s for playback\n", frSetup->fileName);
   err = tmolFreadInstanceSetup(readerInstance, frSetup);
   tmAssert((err == TMLIBAPP_OK), err);

   /* setup audio renderer */
   arSetup->defaultSetup->inputDescriptors[0] = iod;
   arSetup->defaultSetup->errorFunc = arend_error_func;
   arSetup->maxBufferSize = 2 * sizeof(Int16) * BUFSIZE;
   err = tmolArendAOInstanceSetup(arendInstance, arSetup);
   tmAssert((err == TMLIBAPP_OK), err);

/* now everything is ready:  Start the renderer */
   err = tmolArendAOStart(arendInstance);
   tmAssert((err == TMLIBAPP_OK), err);
   err = tmolFreadStart(readerInstance);
   tmAssert((err == TMLIBAPP_OK), err);

   printf("file %s playing as stereo audio. \n", frSetup->fileName);
   printf("Press return to stop\n");
   gets(ins);

/* Stop the File everything. */
   err = tmolFreadStop(readerInstance);
   tmAssert((err == TMLIBAPP_OK), err);
   err = tmolArendAOStop(arendInstance);
   tmAssert((err == TMLIBAPP_OK), err);
   printf("All stopped.\n");
   err = tsaDefaultInOutDescriptorDestroy(iod);
   tmAssert((err == TMLIBAPP_OK), err);
   err = tmolFreadClose(readerInstance);
   tmAssert((err == TMLIBAPP_OK), err);
   err = tmolArendAOClose(arendInstance);
   tmAssert((err == TMLIBAPP_OK), err);

   return;
}
```

## Check Capabilities

```
/* find out what formats are supported */
   tmolFreadGetCapabilities(&frCaps);
   tmolArendAOGetCapabilities(&arCaps);
   if (!(paf->dataSubtype & apfStereo16)) {
      printf("Stereo audio playback not supported on this board.\n");
      return;
   }
```

The capabilities function allows you to find out what formats are supported by the system. The capabilities of each component are required for setup. Hence these calls are made at the start of the program. Ultimately, the board support package is responsible for setting the capabilities of the audio system. The audio formats returned by the renderer are retrieved through the board support package (see “Board Support Package” starting on page -26).

## Open the Components:

```
   err = tmolFreadOpen(&readerInstance);
   tmAssert((err == TMLIBAPP_OK), err);
   err = tmolFreadGetInstanceSetup(readerInstance, &frSetup );
   tmAssert((err == TMLIBAPP_OK), err);

   err = tmolArendAOOpen(&arendInstance);
   tmAssert((err == TMLIBAPP_OK), err);
    err = tmolArendAOGetInstanceSetup(arendInstance, &arSetup );
   tmAssert((err == TMLIBAPP_OK), err);
```

Each component that will be used must be opened. This creates an instance of the component for you to use. The GetCapabilities function is called to retrieve a setup structure that has been initialized to default values. Notice the use of tmAssert. Like the ANSI **assert**, **tmAssert** will halt the program and print the file name and line number on an error condition. In addition, tmAssert() prints the error code, and it prints it all both to STDOUT and to the DP buffer. This assert mechanism is used liberally throughout TM audio code. It is invaluable in the identification of programming errors. And when the program is running, it is easy to turn off the tmAssert checking. Compilation with the flag “-DNO_DEBUG” removes all of the assertion checking. In this way, the assert checking provides strong error checking when appropriate, and it has no run time impact when the code is released.

## Make the Connection Between Each Pair of Components:

```
/* assemble audio format */
   audioFormat.size         = sizeof(tmAudioFormat_t);
   audioFormat.hash         = audioFormat.referenceCount = 0;
   audioFormat.dataClass    = avdcAudio;
   audioFormat.dataType     = atfLinearPCM;
   audioFormat.dataSubtype  = apfStereo16;
   audioFormat.description  = 16;
   audioFormat.sampleRate   = sRate;
/* create an InOutDescriptor */
   iodSetup.format          = (ptmAvFormat_t)&audioFormat;
   iodSetup.flags           = tsaIODescSetupFlagCacheMalloc;
   iodSetup.fullQName       = "full";
   iodSetup.emptyQName      = "mpty";
   iodSetup.queueFlags      = tmosQueueFlagsStandard;
   iodSetup.senderCap       = frCaps->defaultCapabilities;
   iodSetup.receiverCap     = arCaps->defaultCapabilities;
   iodSetup.senderIndex     = 0;
   iodSetup.receiverIndex   = 0;
   iodSetup.packetBase      = 0;
   iodSetup.numberOfPackets = NUMBER_OF_PACKETS;
   iodSetup.numberOfBuffers = 1;
   iodSetup.bufSize[0]      = 2 * sizeof(Int16) * BUFSIZE;
   err = tsaDefaultInOutDescriptorCreate(&iod, &iodSetup);
   tmAssert((err == TMLIBAPP_OK), err);
```

Each pair of TSSA components are connected by a structure called an InOutDescriptor. The function tsaDefaultInOutDescriptorCreate() is used to create one of these connections. The parameters that must be specified are illustrated here. In this example, a valid format structure is passed in when the connection is created. It is also possible to pass in Null. The format can be specified later using the tsaDefaultInstallFormat() command, or it can even be determined after the receiving component has started. This might be more convenient when using a decoder that finds the format in the data stream only after it has decoded some data. In this case, the format is passed in the data packet that travels through the queue inside of the InOutDescriptor.

The CreateInOutDescriptor function can also create the data packets that are used to stream data between the file reader and the audio renderer. These are initially placed in the empty queue. Setting the numberOfPackets field to zero will bypass this step, if you have some special reason to create your own packets. This code illustrates a fairly typical approach to the problem.

## Setup the File Reader

```
    frSetup->defaultSetup->outputDescriptors[0] = iod;
    frSetup->defaultSetup->priority = READER_PRIORITY;
    frSetup->fileName               = fileName;
    printf("Opening %s for playback\n", frSetup->fileName);
    err = tmolFreadInstanceSetup(readerInstance, frSetup);
    tmAssert((err == TMLIBAPP_OK), err);
```

The file reader is a TSSA component that provides a streaming interface to a file. It takes packets from its empty queue, reads from the disk to fill them, and then places the packets in its full queue. As a default, the file reader loops back to the beginning when it reaches the end of the file. More information about the file reader can be found in Chapter 1, “File Reader (Fread) API,” of *Software Library APIs*.

Given the already initialized file reader setup structure that was retrieved after open, the file reader is very simple to setup. A file name is clearly required. The InOutDescriptor is required. And a priority is assigned for the pSOS task that will be created.

The amount of data read in each packet is determined by the bufSize field in the header of each packet. This was initialized when the packets were created and the memory was allocated by **tsaDefaultInOutDescriptorCreate**.

## Setup the Audio Renderer

```
    arSetup->defaultSetup->inputDescriptors[0] = iod;
    arSetup->defaultSetup->errorFunc = arend_error_func;
    arSetup->maxBufferSize = 2 * sizeof(Int16) * BUFSIZE;
    err = tmolArendAOInstanceSetup(arendInstance, arSetup);
    tmAssert((err == TMLIBAPP_OK), err);
```

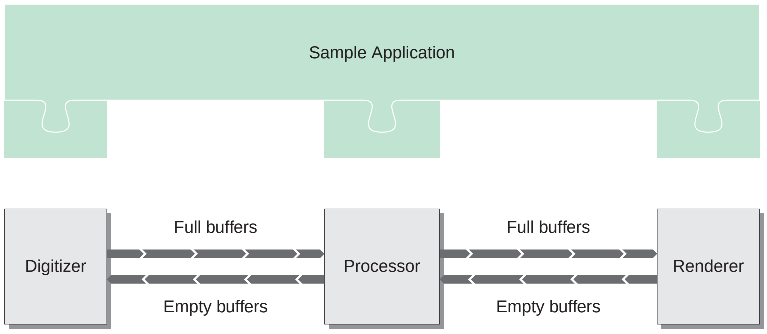
The audio renderer is implemented as an interrupt service routine. It is not a task. Like the reader, a partially initialized setup structure was obtained after the component was opened. The user must specify an InOutDescriptor, and a maximum buffer size. The error reporting function is optional. The format of the audio data stream is specified as part of the InOutDescriptor. After the call to **tmolArendInstanceSetup**, we are ready for start.

## Start

```
err = tmolFreadStart(readerInstance);
err = tmolArendAOStart(arendInstance);

printf("file %s playing as stereo audio.\n", frSetup->fileName);
printf("Press return to stop\n");
gets(ins);
```

The calls to the start functions (**tmolArendAOStart**, and **tmolFreadStart**) cause these two independent components to begin exchanging data. The audio renderer runs in an interrupt service routine. Under current pSOS rules, this uses the stack of the currently running task. The file reader is started as an autonomous task. Since buffers start in the empty queue, the file reader will immediately begin to fill these buffers, and packets will bunch up in the full queue. The audio renderer will be activated after each buffer has played. If these buffers contain 256 samples of stereo audio (1024 bytes), and the sample rate is 44100, the audio renderer will request a new packet every 5.8ms. The renderer requests a new packet from the full queue. In steady state operation, it also places the previous packet in the empty queue. Since the reader task is blocked waiting for a empty buffer, the reader is now ready to run and the cycle can continue.



QuickTime
DATPULL7.MOV

**Figure 1** Sample Audio Application

The **printf** and **gets** provide a simple and convenient development interface. Since this code is in a thread separate from the reader and the renderer, the fact that this thread is blocked has no effect on the other threads.

### Stop and Shutdown

```
    err = tmolFreadStop(readerInstance);
    tmAssert((err == TMLIBAPP_OK), err);
    err = tmolArendAOStop(arendInstance);
    tmAssert((err == TMLIBAPP_OK), err);
    printf("All stopped.\n");
    err = tsaDefaultInOutDescriptorDestroy(iod);
    tmAssert((err == TMLIBAPP_OK), err);
    err = tmolFreadClose(readerInstance);
    tmAssert((err == TMLIBAPP_OK), err);
    err = tmolArendAOClose(arendInstance);
    tmAssert((err == TMLIBAPP_OK), err);
```

When it is time to stop the process, the stop functions are called. The operation of each stop function is synchronous; that is, the stop function will not return until the component being stopped has actually completed its work. Under TSSA, stop means "return all your memory and exit your processing loop." Hence at the end of the stop procedure, all of the packets should be returned to the queues.

### Advanced Features

The audio renderer is a reasonably mature interface. It supports the basic features well, and it also provides some advanced features. One of these is the progress callback function. The progress callback function can be called at every interrupt service routine. This can be used to implement synchronization functions like that required to lock the output to a digital audio input.

Another advanced feature of the audio renderer is its handling of time-stamped packets. If the renderer is set up with a clock reference, and if its packets are time-stamped, the renderer will attempt to present these packets at the correct time. If the packet arrives too early, the renderer will hold onto it until its presentation time arrives. If it is too late, the packet will be returned immediately so as to catch up. This mechanism can be used to implement AV ("lip") sync. It assumes that once sync is achieved, the audio and video will remain in sync. If that is not the case, then the DDS should be used to vary the audio clock so as to achieve long term sync.

## Audio Digitizer

The audio digitizer is an interface to audio input. Like all TSSA components, a section of the API reference manual is devoted to it. Some example programs such as exolAIO are provided as well. The following code illustrates the basic operation of the audio digitizer:

```
/* Get Capabilities */
   rval = tmolAdigAIGetCapabilities(&AdigAICap);
   rval = tmolArendAOGetCapabilities(&ArendAOCap);

/* Open components */
   rval = tmolAdigAIOpen(&digitizerInstance);
   rval = tmolArendAOOpen(&arendInstance);
```

```
/* Get setup variables */
   rval = tmolAdigAIGetInstanceSetup(digitizerInstance, &digitizerSetup);
   rval = tmolArendAOGetInstanceSetup(arendInstance, &arendSetup);

/* create the I/O descriptor to connect components */
   descriptorSetup.format          = (ptmAvFormat_t)&audioFormat;
   descriptorSetup.flags           = tsaIODescSetupFlagCacheMalloc;
   descriptorSetup.fullQName       = "AIOQ";
   descriptorSetup.emptyQName      = "AOIQ";
   descriptorSetup.queueFlags      = 0;
   descriptorSetup.senderCap       = AdigAICap->defaultCapabilities;
   descriptorSetup.receiverCap     = ArendAOCap->defaultCapabilities;
   descriptorSetup.senderIndex     = 0;
   descriptorSetup.receiverIndex   = 0;
   descriptorSetup.packetBase      = 0x100;
   descriptorSetup.numberOfPackets = MAX_PACKETS;
   descriptorSetup.numberOfBuffers = 1;
   descriptorSetup.bufSize[0]      = bytesPerPacket;
   rval = tsaDefaultInOutDescriptorCreate(&iod, &descriptorSetup))

/* setup components */
   digitizerSetup->defaultSetup->errorFunc = digitizer_error_func;
   digitizerSetup->defaultSetup->outputDescriptors[0] = iod;
   rval = tmolAdigAIInstanceSetup(digitizerInstance, digitizerSetup);
   arendSetup->defaultSetup->inputDescriptors[0] = iod;
   arendSetup->defaultSetup->errorFunc = renderer_error_func;
   arendSetup->maxBufferSize = bytesPerPacket;
   rval = tmolArendAOInstanceSetup(arendInstance, arendSetup);

/* now everything is ready:  Start  */
   rval = tmolArendAOStart(arendInstance);
   rval = tmolAdigAIStart(digitizerInstance);
   printf("Press return to stop\n");
   gets(ins);

/* Stop everything. */
   rval = tmolAdigAIStop(digitizerInstance);
   rval = tmolArendAOStop(arendInstance);
   rval = tmolAdigAIClose(digitizerInstance);
   rval = tmolArendAOClose(arendInstance);
   rval = tsaDefaultInOutDescriptorDestroy(iod);
```

You can see the similarity to the setup of other TSSA components. The sequence of **Open**, **GetInstanceSetup**, **InOutDescriptorCreate**, **InstanceSetup**, **Start** is very common. Like the audio renderer, the audio digitizer runs in an interrupt service routine.

One interesting feature of the audio digitizer is its second output. The digitizer has two outputs. This allows the output to be simultaneously routed to a file writer and to the audio renderer as a monitor.

## CopyAudio Example

The copyAudio example program connects the audio digitizer to a simple data copier and through to the audio renderer. This is can easily serve as a starting point for the development of new TriMedia audio modules.

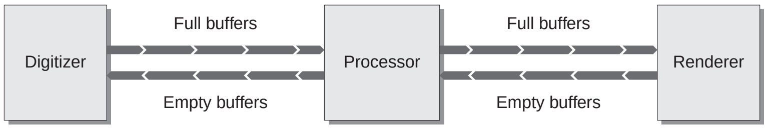


**Figure 2** Example of the flow of a simple audio copy

As you might guess from looking at the picture, this will consist of code to create the modules, create the queues, and connect the modules.

**Note**
To get a good understanding of the following example program, it is recommended that you familiarize yourself with *Book 3, Software Architecture, Part A*.

### Create the Components:

```
tmolAdigAIOpen(&digitizerInstance);
tmolCopyIOOpen(&copyInstance);
tmolArendAOOpen(&arendInstance);
```

Of course the return values must be checked.

### Getting Capabilities

```
if(rval = tmolAdigAIGetCapabilities(&digitizerCap)) {
   printf("Error in tmolAdigAIGetCapabilities: 0x%x\n",rval);
   tmosExit(-1);
}
if(rval = tmolCopyIOGetCapabilities(&copyCap)) {
   printf("Error in tmolCopyIOGetCapabilities: 0x%x\n",rval);
   tmosExit(-1);
}
if(rval = tmolArendAOGetCapabilities(&arendCap)) {
   printf("Error in tmolArendAOGetCapabilities: 0x%x\n",rval);
   tmosExit(-1);
}
bytesPerPacket = BUFSIZE * 2 * sizeof(Int16);
```

## Setting up iosetups

```
iosetup1.format          = (ptmAvFormat_t)&aFormat;
iosetup1.flags           = tsaIODescSetupFlagCacheMalloc;
iosetup1.fullQName       = "digF";
iosetup1.emptyQName      = "digE";
iosetup1.queueFlags      = 0;
iosetup1.senderCap       = digitizerCap->defaultCapabilities;
iosetup1.receiverCap     = copyCap->defaultCapabilities;
iosetup1.senderIndex     = 0;
iosetup1.receiverIndex   = 0;
iosetup1.packetBase      = 0x100;
iosetup1.numberOfPackets = MAX_PACKETS;
iosetup1.numberOfBuffers = 1;
iosetup1.bufSize[0]      = bytesPerPacket;

iosetup2.format          = (ptmAvFormat_t)&aFormat;
iosetup2.flags           = tsaIODescSetupFlagCacheMalloc;
iosetup2.fullQName       = "renF";
iosetup2.emptyQName      = "renE";
iosetup2.queueFlags      = 0;
iosetup2.senderCap       = copyCap->defaultCapabilities;
iosetup2.receiverCap     = arendCap->defaultCapabilities;
iosetup2.senderIndex     = 0;
iosetup2.receiverIndex   = 0;
iosetup2.packetBase      = 0x200;
iosetup2.numberOfPackets = MAX_PACKETS;
iosetup2.numberOfBuffers = 1;
iosetup2.bufSize[0]      = bytesPerPacket;
```

## Creating InOutDescriptors

```
if(rval = tsaDefaultInOutDescriptorCreate(&iodesc1, &iosetup1)) {
   printf("Error in tsaDefaultInOutDescriptorCreate: 0x%x\n",rval);
   tmosExit(-1);
}
if(rval = tsaDefaultInOutDescriptorCreate(&iodesc2, &iosetup2)) {
   printf("Error in tsaDefaultInOutDescriptorCreate: 0x%x\n",rval);
   tmosExit(-1);
}
```

## Setting up audio digitizer

```
rval = tmolAdigAIOpen(&digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolAdigAIGetInstanceSetup(digitizerInstance,&digitizerSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
```

## Setting up first output only

```
digitizerSetup->defaultSetup->outputDescriptors[ADIGAI_MAIN_CHANNEL_OUTPUT]
                                                                    = iodesc1;
rval = tmolAdigAIInstanceSetup(digitizerInstance, digitizerSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
```

## Setting up copy component

```
rval = tmolCopyIOOpen(&copyInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolCopyIOGetInstanceSetup(copyInstance, &copySetup);
tmAssert((rval == TMLIBAPP_OK), rval);
copySetup->defaultSetup->inputDescriptors  [COPYIO_MAIN_INPUT ] = iodesc1;
copySetup->defaultSetup->outputDescriptors [COPYIO_MAIN_OUTPUT] = iodesc2;
rval = tmolCopyIOInstanceSetup(copyInstance, copySetup);
tmAssert((rval == TMLIBAPP_OK), rval);
```

## Opening and setting up audio renderer

```
rval = tmolArendAOOpen(&arendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolArendAOGetInstanceSetup(arendInstance, &arendSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
```

## Initializing Audio Renderer

```
arendSetup->defaultSetup->inputDescriptors[ARENDAO_MAIN_INPUT] = iodesc2;
arendSetup->defaultSetup->errorFunc = renderer_error_func;
arendSetup->operationalMode = AR_MODE_CONSERVATIVE;
arendSetup->maxBufferSize = bytesPerPacket;
rval = tmolArendAOInstanceSetup(arendInstance, arendSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
```

Now that everything is ready, start them all:

```
DP(("\nStarting Audio Digitizer, Renderer, and file writer\n"));
rval = tmolArendAOStart(arendInstance);
   tmAssert((rval == TMLIBAPP_OK),rval);
rval = tmolCopyIOStart(copyInstance);
   tmAssert((rval == TMLIBAPP_OK),rval);
rval = tmolAdigAIStart(digitizerInstance);
   tmAssert((rval == TMLIBAPP_OK),rval);
```

## Waiting for the user to press RETURN to exit the program

```
printf("Audio Copy demo started.  Press return to exit.  \n");
gets(ins);
printf("Stopping all:\n");
rval= tmolAdigAIStop(digitizerInstance); tmAssert((rval==TMLIBAPP_OK),rval);
rval= tmolCopyIOStop(copyInstance);      tmAssert((rval==TMLIBAPP_OK),rval);
rval= tmolArendAOStop(arendInstance);    tmAssert((rval==TMLIBAPP_OK),rval);
rval= tmolAdigAIClose(digitizerInstance);tmAssert((rval==TMLIBAPP_OK),rval);
rval= tmolArendAOClose(arendInstance);   tmAssert((rval==TMLIBAPP_OK),rval);
rval= tmolCopyIOClose(copyInstance);     tmAssert((rval==TMLIBAPP_OK),rval);
```

### Destroying InOutDescriptors

```
if(rval = tsaDefaultInOutDescriptorDestroy(iodesc1)) {
   printf("Error in tsaDefaultInOutDescriptorDestroy: 0x%x\n",rval);
   tmosExit(-1);
}
if(rval = tsaDefaultInOutDescriptorDestroy(iodesc2)) {
   printf("Error in tsaDefaultInOutDescriptorDestroy: 0x%x\n",rval);
   tmosExit(-1);
}

DP(("Demo Complete.\n"));
printf("\nDemo Complete.\n");
```

### Exit the program

```
    tmosExit(0);
```

This approach will get your component up and running quickly. But as your component becomes more mature, you will want to adopt more of the TSA conventions that are illustrated in the “Audio Mixer” section.

### Running The CopyAudio program

1. Copy the CopyAudio program in your editor file.

   **Note**
   For more details about how to run a TriMedia application, refer to Chapter 1 of Getting Started with Philips TriMedia and Chapter 1, *Compiling TriMedia Applications,* of the Cookbook .

2. You may have some difficulties in compiling the program. If this is the case, make sure of the following:
   — The \bsp\configs includes all the necessary extension: (will be copied).
   — The include file includes all the necessary header files: (will be copied).

## Audio Mixer

■ The example program known as exolAmixSimple demonstrates the use of a simple audio mixer. The simple mixer is supplied complete with source. Out of the box, it accepts three stereo inputs and mixes them into one stereo output. This simple audio mixer demonstrates the concepts involved in the construction of a mixer. It is a simplified version of the mixer used with the TriMedia Digital Television (DTV) system. The source for the library illustrates several important concepts:
   — The mixer supports a configuration function with a queued interface.
   — The mixer demonstrates how to handle multiple input pins. The principles are similar for multiple output pins.

— The mixer separates the tmal and tmol layers using a subdirectory. This is done to make it easier to isolate the valuable intellectual property that exists in your code at the AL layer. It is common practice to guard the AL layer source. By making the OL layer source available, it is possible for a client to change the operating system without accessing your private source.

### Audio Decoders

A number of audio decoders are available for use with TriMedia. These include decoders for Dolby AC3 and Dolby ProLogic. Each of these are delivered as TSSA-compatible modules. Since a separate licensing fee is required for these decoders, you are advised to contact your TriMedia sales representative for more information. Code is also available to decode MPEG 1 layer 2 audio, and G.723, although it is not packaged as a TSSA module.

## Audio Device Library

If for some reason the TSSA audio interface is not appropriate, a lower level of access is available. It is this "device library" interface that is used to construct the audio renderer and the audio digitizer. The TSSA interface solves many problems that have deliberately been left unaddressed at the device library level. But of course, there are other ways to solve the same problems.

### Audio Hardware Overview

The TriMedia Audio-In unit connects to an off-chip stereo analog-to-digital (A/D) converter subsystem through a flexible bit-serial bus. It provides all signals needed to interface to high-quality, low-cost oversampling (analog-to-digital) A/D converters, including a precisely programmable oversampling A/D system clock.

The TriMedia Audio-Out unit connects to an off-chip stereo digital-to-analog (D/A) converter subsystem through a flexible bit-serial interface. It provides an interface to high-quality, low-cost oversampling D/A converters and a precisely programmable oversampling D/A system clock.

The Audio-In /Audio-Out unit implements a double-buffering scheme, ensuring that no samples are lost even if the DSPCPU is highly loaded and slow to respond to interrupts.

The Audio-In /Audio-Out unit is reset by writing a **0x80000000** to the **AI_CONTROL/ AO_CONTROL**) register. This disables capture/transmission by setting the **CAP_ENABLE** / **TRANS_ENABLE**) flag to 0, and makes **buffer1** the active buffer by setting **BUF1_ACTIVE** flag to 1.

## Capture/Transmission by DSPCPU

1. The DSPCPU initiates capture/transmission by providing two empty/full buffers and putting their base addresses and sizes in the **BASE**$_n$ and **COUNT**$_n$/**SIZE**$_n$ registers. It does so by writing a base address and size to MMIO control fields.
2. After two valid local memory buffers are assigned, capture/transmission is enabled by setting **CAP_ENABLE**/**TRANS_ENABLE** to 1.
3. The Audio-In /Audio-Out unit hardware then fills/empties **buffer1** by reading input/transmitting output samples. After **buffer1** fills/empties, **BUF1_FULL/BUF1_EMPTY** is asserted and capture/transmission continues without interruption in **buffer2**.
4. Before **buffer2** fills up, the DSPCPU must assign a new, empty/full buffer to **BASE1**, **COUNT1**/**SIZE1**, and perform an **ACK1**. **BUF2_FULL**/**BUF2_EMPTY** is asserted when **buffer2** fills up/empties, and capture/transmission continues in/from the new **buffer1**, and so forth.
5. Upon receipt of an ACK, the Audio-In /Audio-Out hardware removes the interrupt line assertion at the next DSPCPU clock edge. Refer to the interrupt controller documentation for details about interrupt handler programming.

In normal operation, the DSPCPU and the Audio-In /Audio-Out hardware continuously exchange buffers without losing a sample.

However, timing is important in the Audio-In unit. If, for example, the DSPCPU fails to provide a new buffer in time, the **OVERRUN** error flag is raised, causing a temporary halt to input sampling. Sampling resumes as soon as the DSPCPU makes one or more new buffers available through an **ACK1** or **ACK2** operation.

Timing is important in the Audio-Out unit, as well. If, for example, the DSPCPU fails to provide a new buffer in time, the **UNDERRUN** error flag is raised, and the last valid sample or sample pair is repeated until a new buffer of data is assigned by **ACK1** or **ACK2**.

The TriMedia Audio-In/Out APIs provide the necessary interface for audio applications to access the TriMedia Audio-In/Out unit hardware.

## Using the TriMedia Audio-In/Audio-Out API

The functions provided in the TriMedia Audio-In/Audio-Out API enable you to access both the Audio-In and Audio-Out hardware units of TriMedia. The Audio-In/Audio-Out device library provides functions to control audio coders-encoders (codecs) attached to the TriMedia processor, as well as support for the audio mixer and other audio subsystems.

The interface provided by the Audio-In/Audio-Out device library is simple to use. To access the Audio-In or Audio-Out unit, the application program first opens the unit and sets a few parameters, and then initiates capturing or transmission by removing the pause condition. The audio is then serviced by interrupts. After the audio is running, its volume, sample rate, and input selection are controlled by the APIs provided in the Audio-In/Audio-Out device library.

### Guidelines for Use of the Audio-In/Audio-Out APIs

General guidelines for using the TriMedia Audio-In/Audio-Out APIs are as follows:

- Include the <tm1/tmAI.h> and <tm1/tmAO.h> header files.
- Use the archive version (libdev.a), rather than building the library yourself. (The Audio-In/Audio-Out device library is archived in libdev.a.)

  The source for the Audio-In/Audio-Out device library is included in the TriMedia Compilation System (TCS). This makes it easier to incorporate new versions of the library as they become available.
- Pass the specific owner ID when making subsequent calls.

  The Audio-In/Audio-Out device library operates as an exclusive device driver, and, as such, can service only one task at a time. This is enforced through the owner field of the control data structure, which is returned by all the initialization functions.
- Check the error values returned by the initialization functions. Most of the Audio-In/Audio-Out device library functions return zero on success, or nonzero error codes. Many functions check and report the use of sizes and alignments that the hardware cannot support.

### Restrictions

Because of hardware or software limitations, the Audio-In/Audio-Out device library has the following restrictions:

- The buffers must be 64-byte aligned, and buffer sizes must be a multiple of 64 samples.
- Calculation of the sample rate is based on the TriMedia cycle clock. The software gets its definition of this clock from the tmman.ini file residing in the current directory. You must ensure the value of tmman.ini matches your hardware.
- When setting sample rates, consider that the value for the DDS control register is computed in 32-bit math. This might lead to inaccuracies because of truncation. The problem will be fixed in future releases.

## Demonstration Programs

Included with the Audio-In/Audio-Out device library are six demonstration programs:

- fplay
- fplay6
- sine

- sthru
- avio
- patest

If you want to develop audio applications for TriMedia, you can use these demonstration programs to gain an understanding of how to use Audio-In/Audio-Out device library APIs within your applications.

**IMPORTANT**
You will achieve a greater level of compatibility with other TriMedia software modules through the use of the TSSA audio interface.

## Playing an Audio File

The following example demonstrates the role Audio-Out APIs play in an audio file by using the Audio-Out unit. The code is taken from the fplay demonstration program that is provided with the Audio-In/Audio-Out device library.

```
static void
fPlay(char *waveFile, float srate){
   aoInstanceSetup_t  ao;
   FILE              *fp;
   Int                instance, i;
   char               ins[80];

   samples = (int *) malloc(MAX_SAMPLE_SIZE * 4);
   if (!samples) {
      printf("FATAL ERROR:  Error getting sample memory\n");
      exit(1);
   }

   printf("loading sound file %s...\n", waveFile);
   fp = fopen(waveFile, "rb");
   if (!fp) {
      printf("FATAL ERROR:  Failed to open sound file.\n");
      exit(2);
   }
   sample_bytes = fread(samples, 1, MAX_SAMPLE_SIZE, fp);
   printf("sample size is %d bytes.\n", sample_bytes);
   fclose(fp);

   pbuf1 = (int *) (((unsigned long) buf1 + 63) & ~63U);
   pbuf2 = (int *) (((unsigned long) buf2 + 63) & ~63U);

   memset(pbuf1, 0, BUF_SIZE * 4);
   memset(pbuf2, 0, BUF_SIZE * 4);

   for (i = 0; i < BUF_SIZE; i += 16) {
      _cache_copyback(pbuf1, BUF_SIZE);
      _cache_copyback(pbuf2, BUF_SIZE);
   }

   ao.isr = fPlayISR;
   ao.interruptPriority = intPRIO_3;
   ao.audioTypeFormat = atfLinearPCM;
   ao.audioSubtypeFormat = apfStereo16;
   ao.srate = srate;
```

```
    ao.size = BUF_SIZE;
    ao.base1 = pbuf1;
    ao.base2 = pbuf2;
    ao.underrunEnable = True;
    ao.hbeEnable = True;
    ao.buf1emptyEnable = True;
    ao.buf2emptyEnable = True;

    LIBDEV(aoOpen(&instance));
    LIBDEV(aoInstanceSetup(instance, &ao));

    aoEnableLITTLE_ENDIAN();

    LIBDEV(aoStart(instance));

    printf("wave file playing:  Press return to stop.\n");
    gets(ins);

    LIBDEV(aoStop(instance));
    LIBDEV(aoClose(instance));

    exit(0);
}
```

Before initializing the audio output hardware, the fplay demonstration program must do the following:

- Align the buffers on a 64-byte boundary.
- Call the **_cache_copyback** function to ensure cache coherency. This is done because the Audio hardware reads from SDRAM and not from Cache. The **_cache_copyback** function uses an optimized algorithm to flush the cache.
- Set the audio out parameters. The Interrupt Service Routine (ISR) pointer is set to **fPlayISR** (see the following example).
- The fplay demonstration program then initializes the Audio-Out hardware by calling the **aoOpen** function, which assigns the audio control block to the owner for exclusive use. **LIBDEV** checks the return value of the **aoOpen** function before proceeding.

The format and the interrupt parameters are initialized from the values in AO. The endianness is set to little endian to conform to the file format.

The procedure halts until a console line is read. Wave file playback is interrupt driven. After the input, the AO unit is stopped and closed.

## Interrupt Routine fplayISR

The following is a description of the interrupt routine **fplayISR**, which is followed by code excerpts that illustrate sequential operations.

The pragma tells the compiler to save and restore the interrupt state. The routine first checks for data underrun and highway bandwidth error conditions and acknowledges them.

There are two Audio Out buffers, with empty status bits for each. If the second is empty, it is filled with the data in “sample” (a circular buffer). The data read is copied back to memory and the interrupt is acknowledged. If buffer 1 is empty, it is handled in the same manner as buffer 2.

The pragma at the end of the function forces a decision tree jump. This is to allow sufficient time between the acknowledgment and the return from interrupt.

The code for the interrupt routine **fplayISR** is shown below.

```
static void
fPlayISR(void){
#pragma TCS_handler

    int         i;
    UInt        stat = MMIO(AO_STATUS);
```

1. Check for underrun and highway bandwidth errors.

```
if( aoUNDERRUN(stat)) aoAckACK_UDR();
if( aoHBE     (stat)) aoAckACK_HBE();
```

2. Next, it copies data to AO buffer 2, if it is empty, and it resets the pointer if it is at the end of the buffer.

```
if( aoBUF2_EMPTY(stat) ){
   for (i = 0; i < BUF_SIZE; i++) {
      pbuf2[i] = samples[sample_pos];
      if (sample_pos++ >= (sample_bytes >> 2))
      sample_pos = 0;
   }
```

3. Next, it forces the cache to write data to memory and it acknowledges the interrupt.

```
for( i = 0; i < BUF_SIZE; i += 16 )
   _cache_copyback(pbuf2, BUF_SIZE);
   aoAckACK2();
}
```

4. **fplayISR** uses the same code for AO buffer 1 that it used with buffer 2:

```
if( aoBUF1_EMPTY(stat) ){
for( i = 0; i < BUF_SIZE; i++ ){
   pbuf1[i] = samples[sample_pos];
```

5. Next, it resets the pointer if it is at the end of the circular buffer.

```
   if (sample_pos++ >= (sample_bytes >> 2)) sample_pos = 0;
}
for (i = 0; i < BUF_SIZE; i += 16)
   _cache_copyback(pbuf1, BUF_SIZE);
   aoAckACK1();
}
#pragma TCS_break_dtree
}
```

## Recording an Audio File

The following example demonstrates the use of Audio-In APIs to create an audio file by reading audio data from the Audio-In unit. The code is taken from the sthru demonstration program, which is provided with the Audio-In/Audio-Out device library.

## sthru Demonstration Program

The first part of the sthru demonstration program allocates and clears the capture buffer and the data buffers. (This code is not shown).

On receiving the interrupt, the DSPCPU executes the interrupt service routine **inISR** (see the following example). The interrupt routine first reads the newly captured data from the inactive buffer pointer using **aiGetBase** and then writes a new pointer to the buffer that is ready for capture data using **aiChangeBuffer**. Because the captured data is not cache-coherent, stale data is removed from the buffer using **invalidate**. Finally, the ISR acknowledges the interrupt by clearing the bit in the status register.

```
void sCapture(float srate){
   AUDIO_CB    in_a;
   int         i, j;
   FILE       *fp;
   int         retval;
   int        *p1,*p2;

   ptr = rawPtr;
   capCount = 0;

/* setup control structure */
   in_a.format = AIO_FORMAT_STEREO_16;
   in_a.sRate_hz = srate;
   in_a.size_samples = BUFSIZE;
   in_a.flags = 0;
   in_a.isr = capISR;

   retval = aiOpen(&in_a, &in_owner);
   if( 0 != retval ){
      printf("aiOpen failed with %d.  Aborting...\n", retval);
      return;
   }
   if( 0 != aiSetBufferSize(in_owner, in_a.size_samples) )
      printf("aiSetBufferSize failed (illegal size?)\n");

   p1 = (int *) (((int) ptr + 63) & 0xFFFFFFC0);
   p2 = &p1[BUFSIZE];
   ptr += BUFSIZE;
   if( 0 != aiSetBuffer1Base(in_owner, p1) )
      printf("aiSetBuffer1Base failed (illegal alignment? 0x%x)\n", p1);

   printf("aiSetBuffer2Base failed (illegal alignment? 0x%x)\n", p2);
   printf(
   "\nCapturing %d seconds of audio input..\n", (mallocSize>>2)/ (int)srate);

   aiUnpause(in_owner);

   while (capCount < ((mallocSize>>2)/BUFSIZE -2))
       if( (capCount % 192) == 0) printf("..\n");
   printf("writing data to 'capture.bin'...\n");
   aiPause(in_owner);
   aiClose(in_owner);

   fp = fopen("capture.bin", "wb");
   if (!fp){
      printf("Failed to open capture file.\n");
      return;
   }
   printf("Wrote %d words into capture.bin. \n", fwrite(rawPtr,
      sizeof(int), mallocSize>>2, fp));
   fclose(fp);
   printf("capture Test completed\n");
}
```

## Setting Audio Parameters

After the audio is running (capture or transmission), you can change the volume (left and right gain), sample rate, and input source by using the APIs provided in the Audio-In/Audio-Out device library.

The following examples demonstrate the use of these APIs. All of the code is taken from a demonstration program, which is provided with the Audio-In/Audio-Out device library.

The following code uses the **aiSetSampleRate** and **aoSetSampleRate** APIs to set the Audio-In and Audio-Out sample rates.

```
aoSetSampleRate( out_owner, srate );
aiSetSampleRate( in_owner,  srate );
```

For analog input/output devices (such as the AD1847), both the audio input and audio output are performed by the same chip. Therefore, both the input and output use the same sample rate. In such cases, you can use either function.

Audio-In and Audio-Out each have an instance setup structure. These are initialized with the interrupt parameters and formats.

1. The following code shows the call to the open routines (**aoOpen**, **aiOpen**) to acquire an instance, and the instance setup routines (**aoInstanceSetup**, **aiInstanceSetup**) are then called with the instance value and the appropriate parameters.

   ```
   int main( int argc, char **argv ){
      aoInstanceSetup_t  ao;
      aiInstanceSetup_t  ai;
      char               ins[80];
      int                i;
      int               *buf;
      FILE              *fp;
   ```

   The initialization code, argument checking code, and buffer setup is not shown.

2. The following code sets up the audio formats.

   ```
   /* setup control structure */
      ai.audioTypeFormat = ao.audioTypeFormat = atfLinearPCM;
      if (monoFlag)
         ai.audioSubtypeFormat = ao.audioSubtypeFormat = apfMono16;
      else
         ai.audioSubtypeFormat = ao.audioSubtypeFormat = apfStereo16;
   ```

3. Set up the interrupt service routine and the priority level.

   ```
   ao.isr = outISR;
   ai.isr = inISR;
   ao.interruptPriority = ai.interruptPriority = intPRIO_3;
   ```

4. Set up the sampling rate, and the size and buffer pointers (for AO).

```
ao.srate = ai.srate = sRate;
ao.size  = ai.size  = BUFSIZE;
ao.base1 = b[0];
ao.base2 = b[1];
```

5. Set up the interrupt enable flags for AO.

```
ao.underrunEnable  = True;
ao.hbeEnable       = True;
ao.buf1emptyEnable = True;
ao.buf2emptyEnable = True;
```

6. Setup the buffer pointers and the interrupt enable flags for AI.

```
ai.base1 = b[2];
ai.base2 = b[3];
ai.overrunEnable  = True;
ai.hbeEnable      = True;
ai.buf1fullEnable = True;
ai.buf2fullEnable = True;
```

7. Open AO and AI and configure the device.

```
ERROR_REPORT(aoOpen(&ao_instance));
ERROR_REPORT(aoInstanceSetup(ao_instance, &ao));
ERROR_REPORT(aiOpen(&ai_instance));
ERROR_REPORT(aiInstanceSetup(ai_instance, &ai));
```

8. The input and output volumes (left and right channels) are set in hundredths of DB. A negative value corresponds to attenuation and a positive value to gain. Note that the input volume must be positive or zero and the output volume must be negative or zero. The input is set to the line input of the microphone.

```
aoSetVolume( ao_instance, outputVolume * 100, outputVolume * 100 );
aiSetVolume( ai_instance, inputVolume  * 100, inputVolume  * 100 );
```

9. Set the input to the line or to the mike.

```
if( mode == MODE_MIC ){
      ERROR_REPORT(aiSetInput(ai_instance, aaaMicInput));
   else
      ERROR_REPORT(aiSetInput(ai_instance, aaaLineInput));
   ERROR_REPORT(aiStart(ai_instance));
   ERROR_REPORT(aoStart(ao_instance));
```

10. Pause until the user types the following:

```
<CR>
    printf("Audio Pass Thru is running.  Press return to exit :\n");
```

11. Stop and close AI and AO.

```
ERROR_REPORT(aoStop(ao_instance));
ERROR_REPORT(aiStop(ao_instance));
ERROR_REPORT(aoClose(ai_instance));
ERROR_REPORT(aiClose(ai_instance));
```

12. Write the data to capture.bin (binary mode).

```
if( captureFlag ){
      printf("Writing %d samples of captured data to "
         "capture.bin...\n", CAPSIZE);
      fp = fopen("capture.bin", "wb");
      if( !fp ){
         printf("FATAL ERROR:  capture.bin fopen failed\n");
         exit(2);
      }
      fwrite(capBuffer, 1, CAPSIZE * 4, fp);
      fclose(fp);
      printf("Done!\n");
   }
   exit(0);
}
```

13. After receiving the interrupt, the CPU executes the **inISR** ISR. (See the following example).

    The ISR first determines which buffer is inactive by using the **aoBuf1Active** macro (defined in tmAO.h). It then fills the inactive buffer with new audio data, flushes the cache, and finally acknowledges the interrupt by clearing the bit in the status register.

```
static void
inISR(void){
#pragma TCS_handler

   int     *foo;
   int      i;
   UInt     stat;
```

14. Increment the input buffer pointer modulo 4.

```
   inBuf++;
   inBuf &= 0x3;
```

15. Read the AI status. Check for exceptional conditions.

```
   stat = MMIO(AI_STATUS);
   if( aiOVERRUN(stat) ) aiAckACK_OVR();
   if( aiHBE    (stat) ) aiAckACK_HBE();
```

16. If buffer 2 is full, set foo to the pointer. Switch to the next available buffer.

```
if( aiBUF2_FULL(stat) ){
   foo = (int *) aiGetBASE2();
   aiChangeBuffer2(ai_instance, b[inBuf]);
   aiAckACK2();
}
```

17. If buffer one is full, set foo to the pointer. Switch to the next available buffer. The two buffers should never be full simultaneously.

18. Invalidate any stale data in the cache.

```
if( aiBUF1_FULL(stat) ){
   foo = (int *) aiGetBASE1();
   aiChangeBuffer1(ai_instance, b[inBuf]);
   aiAckACK1();
}
for( i = 0; i < BUFSIZE; i += 16 )
      INVALIDATE((char *) &foo[i], 1);
```

19. Copy the data into the capture buffer.

```
for( i = 0; i < BUFSIZE; i++ ){
   if( capPtr >= CAPSIZE ) break;
   capBuffer[capPtr++] = foo[i];
}
```

There are decision tree breaks previously, so this one is actually unnecessary.

## Board Support Package

The board support package is an integral part of the TriMedia audio system. It is the lowest functional level of the interface. It is at this level that the actual capabilities of the system are determined.

The board support package delivered with the TriMedia developers kit includes support for a number of boards. These include the standard "IREF" board, as well as Philips reference boards for DTV. The board support package detects which board is in use and selects the appropriate function tables to drive that board. This mechanism is explained in some depth in Chapter 19, *TMBoard API*, of Book 5, *System Utilities*.

Some examples of the types of capabilities that can be supported through the board support package are:

- The IREF hardware cannot support simultaneous stereo input and six channel output. This is coded into the board support package.
- The DTV board supports 8 channels of 20-bit audio output. This is done using an external FPGA with the audio clock running at double speed. All of the setup for this configuration is in the board support package.
- The AD1847 on the IREF board supports volume control. This is accessible because it is supported in the board support package.
- The DTV board supports digital audio input. The code to control this resides in the board support package.