Application Report
SPRA802A – February 2003

Writing DSP/BIOS Device Drivers for Block I/O

Shawn Dirksen and Ron Birkett — *Software Development Systems*

ABSTRACT

This application note describes a method for developing block-oriented I/O device drivers for applications that use the DSP/BIOS real-time kernel and includes examples that run with Code Composer Studio v2.1 on the Texas Instruments TMS320C5402 and TMS320C6711 DSP Starter Kits (DSKs). The device driver model presented here has now been superceded with an updated version that supports not only block oriented devices, but also devices such as UARTs, PCI and USB buses and Multimedia cards. Documentation on the updated driver model as well as example drivers and source code can be found in the Device Driver Developer's Kit product now available for download from the TI Developer's Village.

Contents

Figures

Tables

1 Introduction

The drivers described in this application note are intended for use in systems that require frame-based streaming I/O: that is, systems in which the data consists of blocks of data to be processed as a unit with a real-time deadline. Such systems use algorithms that include vocoders and call progress tone detectors and generators, speech recognition, MPEG players, and video processing such as frame- and block-based compression or real-time image analysis. The common element in these applications is that blocks of data are read (or written) periodically from (or to) a data converter in a continuous stream.

First, we look at the requirements for a driver API. This is done by examining the I/O services supported by DSP/BIOS and the threading models with which the I/O services can be used. Then we discuss system requirements such as code and memory overhead. Next, a flexible LIO or “Low-Level I/O” driver API interface is presented along with description of adapters and device controllers. Two implementations are detailed, one using a sample-by-sample interrupt and another using the Enhanced Direct Memory Access or EDMA. Source code examples for both implementations are provided for the TMS320C5402 DSK and the TMS320C6711 DSK. The application examples demonstrate configuration of these drivers. The final section describes performance characteristics and overhead.

2 Device Driver Requirements

2.1 Application Considerations

DSP/BIOS provides two I/O services—pipes (PIP) and streams (SIO)—for implementation of frame-based signal processing systems. Both I/O approaches provide data buffer management for transfers between the application and peripherals and a method to signal program threads when a buffer transfer has completed. (Designers may choose to create their own I/O implementation, but this is not discussed in this application note. Source code examples for custom buffer management implementation can be found in the "rawtest" folder.)

Often program threads are preemptable and are blocked from executing until the required I/O is complete. DSP/BIOS provides two types of preemptable threads: software interrupts (SWIs) and tasks (TSKs). Preemptable threads are required for multi-rate systems such as universal port telecom, and are useful in any system with multiple frame sizes or data rates. PIP objects may be used with SWI or TSK threads. SIO objects may be used only with TSKs.

Since DSP/BIOS provides multiple I/O services and multiple thread types, device drivers written for DSP/BIOS should not constrain users to a particular type of I/O or thread. In order for a driver to be truly universal and to achieve code reuse goals, each buffer management and signaling approach should be supported. Let’s look at the requirements this places on our driver API:

- **Pipes (PIP).** The PIP module statically allocates I/O buffer memory at compile time. Simple allocate and free APIs control which buffer is written to or read from. When a buffer is filled, the writer thread (typically a hardware interrupt) calls PIP_put. In response, the PIP module calls a reader notification function that schedules the data for processing. Typically the reader notification function is a DSP/BIOS kernel call such as SWI_post (when signaling SWI threads). When the reader finishes processing the buffer, it calls PIP_free. In response, the PIP module calls a writer notification function so that the buffer can be reused. Drivers for PIP must use PIP functions such as PIP_put to manipulate the PIP buffers. The driver must also respond to reader and writer notification functions configured for the PIP object.

- **Streams (SIO).** The SIO module is used with TSK threads. It uses dynamically changeable buffer addresses. Typically an application allocates SIO buffers statically or dynamically at start up and reuses the buffers cyclically. However, the SIO API does not require buffers to be used in the same order or buffers to be reused. Drivers we write for SIO must therefore accept any buffer address and not require that buffers be reused as they are in PIP. Internally SIOs use the associated DEV (device) module. SIOs always use semaphores (SEM) to signal the TSK that a transfer has completed. This is in contrast to the lower-level reader/writer notification functions of PIPs.

In sections that follow, we'll see that the driver API we define supports both PIP and SIO buffer management and signaling through a low-level buffer management interface and a callback-signaling interface. The interface is simple and broad enough to support application-specific buffering and signaling in addition to SIO and PIP.

2.2 Device and System Considerations

For a driver API to be universal, it must expose basic peripheral configuration information as well as advanced hardware features not present in all devices. Common basic features of data converters include word size and sample rate. In frame-based systems, data is moved using a DMA channel or autobuffering unit that has reload, or hardware queuing, capability. Advanced features include companding, filtering, and other data preprocessing. The driver API must provide a common interface to such features.

We must also consider system implementation issues such as code size and overhead. If too many features are required, the driver becomes too big for a variety of systems. Therefore the interface should be flexible so that peripherals with simple features and requirements can have small, efficient drivers. By the same token, the API should allow expansion so that a sophisticated device can be used without rewriting significant pieces of an application.

3 Device Driver Model Overview

The DSP/BIOS device driver model consists of two separate pieces: an *adapter* and a *device controller*. Together, these pieces connect hardware devices to threads (SWIs or TSKs).

- **Adapters.** The interface between the application threads and the device controller is provided by an adapter. The adapter communicates directly with a PIP or SIO object and presents a buffer to the device controller. Device-independent issues like buffer management and thread signaling are dealt with by the adapter portion of the device driver. Two LIO adapters are provided: the PLIO adapter is used with the PIP module, and the DLIO adapter is used with the SIO/DEV modules. These adapters provide a common interface to device controllers. These adapters typically need little or no customization.

- **Device controllers.** The interface between an adapter and the hardware is provided by a device controller. The device controller interfaces with the hardware and either fills or empties the buffer presented to it by the adapter. Device controllers are typically small and deal mainly with device-specific issues. A separate device controller must be written for each device. The interface between the device controller and the adapter is designed so that both the PLIO and DLIO adapters can be used with any controller that implements the LIO (Low-level I/O) interface definition.

Adapters are separate from device controllers because adapters use DSP/BIOS function calls that are specific to either the PIP or SIO buffering method. In addition separating the adapter from the device controller minimizes the amount of code that must be written to implement a controller for a new device.

Figure 1 shows the components of a DSP/BIOS device driver.

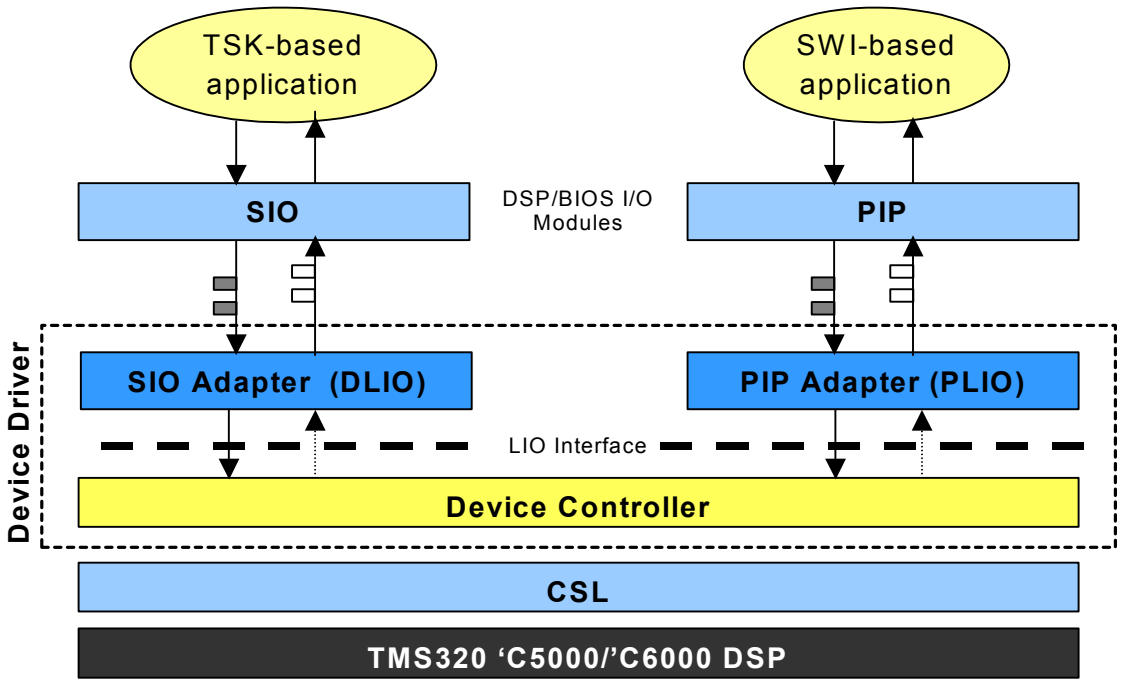


Figure 1. The DSP/BIOS Device Driver Model

3.1 LIO Interface

LIO is a low-level interface between application threads and block-oriented hardware devices. LIO uses a simple buffer management principle and a flexible callback-signaling interface to connect hardware devices to application threads. The LIO interface defines a set of functions and a data structure that must be implemented to create a new LIO device controller.

3.2 LIO Adapters

An LIO adapter obtains a buffer from the application through the buffer manager (PIP or SIO) and presents it to the device controller to be emptied or filled. The adapter recognizes when the controller is finished with the buffer and sends it back to the application through the buffer manager. This is accomplished with a minimal amount of overhead and complexity.

Two basic LIO adapters are provided. The first is the PLIO adapter, which is used with the PIP module. The second is the DLIO adapter, which is used with the SIO/DEV modules. The PIP or SIO object for each adapter is initialized by either the PLIO_new() or SIO_create() function. These functions are called in main() to initialize a data structure for the adapter. This data structure, or instance object, also contains information about the controller to which the adapter presents buffers. This instance object is the glue between the application thread and the adapter, and also between the adapter and the device controller.

3.3 LIO Device Controllers

LIO device controllers form the interface between the adapter and the buffer-oriented hardware. This interface is simple and well-defined so that a controller can be used with either PIP or SIO adapters with minimal overhead. It is also flexible enough to interface to hardware that is simple and hard-wired or complex and programmable. The basic functionality of the controller is signaling and buffer management between the adapter and the hardware. The adapter calls the controller when a new buffer is sent from the application. The controller presents this buffer to the hardware, which is usually represented by an interrupt service routine (ISR). When the ISR is finished with the buffer, it signals the adapter via a "callback" function appropriate for the operation being performed, either receive or transmit. Other capabilities that the controller must provide to the application are initializing the hardware, turning it on and off, and terminating data owned by the device driver. If the controller represents hardware that can be programmed or changed on the fly, an interface is also provided to customize or extend the basic functionality of the controller. These capabilities provide a simple, flexible interface to hardware from an application thread.

4 LIO Interface

The LIO interface defines a set of functions and a data structure that must be implemented to create a new LIO device controller.

The device controller interfaces with the hardware to form the source or destination of data in a system. The device controller initializes the underlying hardware needed by the device driver, manages buffers moving between the application and the hardware, and provides a customizable interface between the application and the hardware. These capabilities are implemented through a set of defined functions and a data structure used to share state information and data buffers.

Since the device-independent issues like buffer management and thread signaling are dealt with by the adapter portion of the device driver, the device controller portion is typically small and deals mainly with device-specific issues.

The functions that make up a controller are listed in Table 1 and described in detail in Appendix A: LIO API Reference, on page 34. They can be divided into three categories: global functions, channel control functions, and ISRs.

Table 1. Device Controller Functions (LIO Interface)

Function Name	Category	Description
<*controller*>_init()	Global function	Initialize controller
<*controller*>_setup()	Global function	Set up controller
open()	Channel control function in LIO_Fxns structure	Open channel
close()	Channel control function in LIO_Fxns structure	Close channel
submit()	Channel control function in LIO_Fxns structure	Submit buffer
cancel()	Channel control function in LIO_Fxns structure	Cancel buffer submission
ctrl()	Channel control function in LIO_Fxns structure	Issue channel-specific command
(driver-dependent)	ISR function	Notify callback function

4.1 Global Functions—init() and setup()

The init() function initializes the controller module. This function might initialize global data structures for the I/O channels of an individual controller. The init() function should be named as shown and have no arguments or return value:

```
Void <controller>_init();
```

The setup() function is used to set parameters for the controller module and perform hardware initialization needed by the controller. The setup() function typically configures and performs hardware initialization using the Chip Support Library (CSL). The setup() function should be named as shown and pass a controller-specific setup structure:

```
Void <controller>_setup(<controller>_Setup *setup);
```

Both of these functions must be called from the application's main() during system initialization. The setup() function must be called after the init() function.

This method works well for static systems that do not change during run-time. The code and data needed for initialization exist in one function that is called only once. Memory needs are reduced by this approach. For example, in a memory-sensitive application, the memory occupied by both the init() and setup() functions can be reused for data. Hardware initialized by the setup() function is owned by the device driver for the life of the application.

If a system needs more flexibility, channel-specific initialization can be performed in the open() function, and the close() function can be used to release resources to the application. In this case, the hardware is owned by the device driver only between open() and close() functions.

With either method, the hardware owned by the device driver should not be used by any other functions in the application. This hardware ownership should be clearly documented by the device controller.

Since the init() and setup() functions touch hardware and the channel objects used by the controller, they should only be called once and before any channel is created or needed by the application. The setup() function should contain logic to ensure that this rule is followed.

4.2 Channel Control Functions

After a controller has been initialized and set up, the application should create instances of the controller (also called channels) by calling PLIO_new() or SIO_create(). Each channel uses its channel object to maintain its state and buffer information.

A LIO_Fxns structure binds the controller to the adapter and completes the device driver. This structure contains five functions that manage the operation of a controller. The open() and close() functions create specific channels. The submit() and cancel() functions control the flow of buffers in these specific channels. Finally, the ctrl() function extends the controller interface to allow for flexibility.

The interface to each of the functions allows interfacing to either a PIP adapter or an SIO adapter. So, when an adapter needs to call a function in the controller, it references this structure within its instance object. Another important piece of the adapter's instance object is a channel-specific object that is initialized by the controller's open function.

4.2.1 open() and close()

When the application calls either PLIO_new() for a PIP adapter or SIO_create() for an SIO adapter, open() is called for the controller that is bound to the adapter that was called. The open() function creates and initializes a specific channel instance of a controller. A channel in this case is a unidirectional flow of buffered data to or from hardware. For example, a codec is represented by an input channel and an output channel. Both of these channels are part of a codec device driver, and the open() function is called when a channel is needed to communicate with the codec. That is, open() is called for both the receive and transmit channels.

The open() function should allocate the appropriate channel depending upon whether the adapter is set up for input or output. The information needed to set up this channel object is passed through arguments to open(). The open() function should also perform any functions the controller author sees as channel-specific. The channel object should contain any variables needed in order to maintain the state of a channel and the data it is consuming. A good example of this type of information might be the buffer count in the controller. The open() function should initialize this variable to zero to indicate an initial state. Any other channel-specific information should be set up by the open() function and stored in the channel object. The open() function should have the following arguments and return value:

```
Ptr open(String name, LIO_Mode mode, Ptr args, LIO_Tcallback cbFxn, Arg cbArg);
```

The close() function is called by the application through SIO_delete() when using an SIO adapter. It should turn off a channel by setting a state variable in the channel object. It should also disable any interrupts that are being used by the channel. This operation allows a channel to gracefully shut down its associated hardware. Note that the PLIO adapter does not call close(). The close() function should have the following argument and return value:

```
Bool close(Ptr chanp);
```

4.2.2 submit() and cancel()

Once a controller has been initialized and a channel has been opened, the application can start sending or receiving data to or from the controller. The submit() function receives buffers from the application and presents them to the underlying hardware. The adapter calls the submit() function when it receives a new buffer from the application. The channel object is used to manage the buffers inside the controller. A pointer to the channel object is passed to submit(), as well as a pointer to the new buffer and the size of the buffer in minimal addressable units (MAUs)[1]. This information should be stored in the channel object. The buffer count should also be updated to indicate that a buffer has been added to the controller. These variables can be used to synchronize and communicate with the ISR. For example, a sample-by-sample controller uses buffer size to tell the ISR how much data to send or receive. The submit() function should have the following arguments and return value:

```
Int submit(Ptr chanp, Ptr buf, Uns nmaus);
```

[1] MAUs for data is different for each processor: 'C6000 is 8-bit byte, 'C5500 is 16-bit byte, and 'C5400 is 16-bit word.

The cancel() function allows an application to terminate the data that is currently owned by a controller. The application can invoke this function by calling SIO_idle() when using an SIO adapter. The cancel() function should signal the underlying hardware to stop sending or receiving data. The PLIO adapter does not call the cancel() function. The cancel() function should have the following argument and return value:

```
Bool cancel(Ptr chanp);
```

4.2.3 ctrl()

The ctrl() function is a way for the controller author to extend the interface to include hardware-specific functionality. If a hardware device provides capabilities that a controller should expose to the application, this function allows the application to pass commands that map to these capabilities down to the hardware. The ctrl() function receives a pointer to the channel object for the channel that the application wants to modify, a command, and an argument to the command. This interface is designed to be flexible in order to meet the needs of many controller writers. This interface should also be clearly documented, as it is an extension of the standard interface. The ctrl() function should have the following arguments and return value:

```
Bool ctrl(Ptr chanp, Uns cmd, Ptr args);
```

4.3 Interrupt Service Routine

The ISR is the device controller function that is actually synchronized to the hardware. It is invoked by hardware through the interrupt vector table and must be enabled in order to be recognized. The ISR actually exchanges data with the hardware. It uses the channel object to share state information with the rest of the controller. For example, when a new buffer is received from the application, the ISR can read this information to know where to put data as it arrives. When the buffer is full of data, the ISR notifies the adapter via a callback function that is contained in the channel object.

4.4 Summary

The device controller interface is a simple yet flexible way to bind an application to a specific piece of hardware. The controller uses three basic types of functions to share data between an adapter and hardware. The initialization and setup functions perform necessary hardware and software initialization. The five channel control functions make up a significant part of the functionality of the device driver. The submit() and ISR functions combine to form the heart of the device driver.

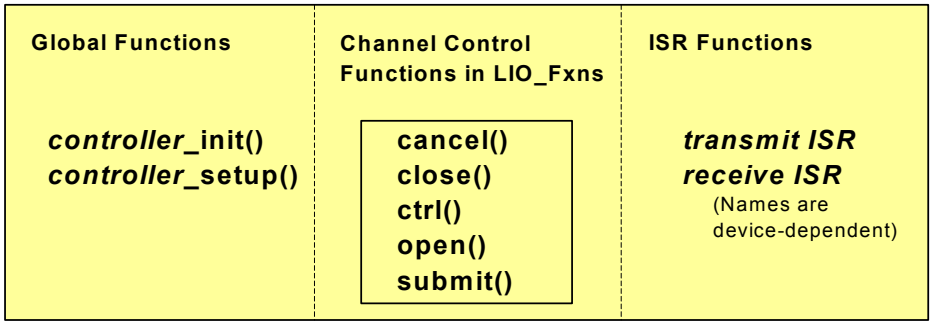


Figure 2. LIO Device Controller Function Summary

5 LIO Adapters

Two LIO adapters are provided to support the I/O services of DSP/BIOS. The first is the PIP adapter (PLIO), which is used with the PIP module. The second is the SIO adapter (DLIO), which is used with the SIO/DEV modules.

5.1 The PIP Adapter (PLIO)

The DSP/BIOS PIP module provides a “data pipe” service. PIPs are designed to manage block I/O. Each pipe object maintains a buffer divided into a fixed number of fixed-length frames. The size and number of frames for a PIP are set in the DSP/BIOS Configuration Tool. Although each frame has a fixed length, the application may put less than a full frame of data into a PIP.

A PIP has two ends. The writer end is where the program writes frames of data. The reader end is where the program reads frames of data. Typically one end is code that invokes an I/O device. Data notification functions are performed to synchronize data transfer. These functions are triggered when a frame of data is read or written to notify the other end of the PIP of the availability of a full or empty frame. A writer gets a frame to put data into by calling the PIP_alloc() function. After data is written to the frame the writer calls PIP_put(). This call results in the notifyReader function being called. When appropriate, the reader calls PIP_get() to retrieve the frame of data and then calls PIP_free() when the data is no longer required. The PIP_free() call triggers the notifyWriter function and the cycle begins again. The notify functions associated with a PIP object are set by the user in the DSP/BIOS Configuration Tool.

The PIP adapter, also referred to as PLIO, is designed to obtain a buffer from the application through the buffer manager and present it to the controller for consumption. The adapter also recognizes when the controller is finished processing the buffer and sends it back to the application through the buffer manager. This communication is accomplished with a minimal amount of overhead and complexity.

The PLIO adapter uses the following basic types of functions:

- **Prime functions.** The PIP buffer manager calls rxPrime and txPrime when the application sends a buffer to the device driver. These functions use DSP/BIOS API calls to obtain a buffer from the buffer manager and present it to the controller. The "prime" functions are the signaling interface between the application and the adapter.
- **Callback functions.** The rxCallback and txCallback functions are the signaling interface between the controller and the adapter. During driver setup, the adapter tells the controller which functions to call when it finishes with the buffer. This callback signals the adapter when a buffer is ready to be sent back to the buffer manager and ultimately, the application.
- **Transfer function.** This function calls the device controller's submit() function. The submit() function of the controller receives a buffer from the adapter and then communicates the new buffer information to the ISR. This communication is done through the channel object.

The PLIO adapter uses these functions to communicate between the application and the controller. This is detailed in Figure 5, which is in Appendix B: PLIO Adapter, on page 39. The PLIO adapter is provided as an adapter to the DSP/BIOS PIP module inside this application note.

5.2 The SIO Adapter (DLIO)

The DSP/BIOS streaming I/O (SIO) module provides a high-level device independent I/O mechanism for use with TSK threads. SIO goes beyond PIP by offering the ability to create new SIO objects at run-time. To provide this ability, SIO has its own device driver model, called DEV. DEV is described in detail in the DSP/BIOS manuals and online help. Writing a DEV is similar to writing an LIO. A small set of device-specific functions, such as open, close, and buffer management, are implemented and accessed by an SIO object through a function table. DEVs are more difficult to write because they require a higher level of DSP/BIOS knowledge. The writer must know the QUE, SEM, and SIO modules in order to implement an SIO DEV driver. A DEV driver can only be used with an SIO and cannot be used with a PIP or on its own.

The SIO adapter, also called DLIO, is designed to easily integrate with the existing DEV module. Communication and synchronization is accomplished with minimal overhead and complexity.

The DLIO adapter uses the following basic types of functions:

- **Callback functions.** The callback functions are the signaling interface between the controller and the adapter. During device driver setup, the adapter tells the controller which functions to call when it finishes with the buffer. This callback signals the adapter when a buffer is ready to be sent back to the buffer manager and ultimately, the application.
- **Transfer function.** This function calls the device controller's submit() function. The submit() function of the controller receives a buffer from the adapter and then communicates the new buffer information to the ISR. This communication is done through the channel object.

The DLIO adapter uses these functions to communicate between the application and the controller. This is detailed in Figure 6, which is in Appendix C: DLIO Adapter, on page 45. The DLIO adapter is provided as an adapter for the DSP/BIOS SIO/DEV modules inside this application note.

6 Writing LIO Device Controllers

Streaming device controllers can be broken down into two different types: sample-by-sample and DMA enabled. A sample-by-sample controller sends or receives a single sample for each hardware interrupt. A DMA enabled controller uses the DMA to send and receive samples and only receives an interrupt at the end of each buffer. After each interrupt, a DMA may have the capability to auto-initialize itself for the next buffer transfer. This capability is extremely powerful and should be used by a controller if the underlying hardware supports it. Using the DMA is preferable in many systems because it decreases the overhead of context switching to and from the ISR, which in turn allows the CPU to do more processing of the data. The differences between these two types of controllers merit a separate discussion for each. The controller function descriptions below make a clear distinction between sample-by-sample controllers and DMA enabled controllers to clarify the purpose of these functions for each type of controller.

6.1 Sample-by-Sample with the McBSP on the C5402 DSK

A sample-by-sample controller uses an ISR to send or receive one sample at a time. The submit() function of the device controller receives a buffer from the adapter. It then communicates the new buffer information with the ISR. This communication is done through the channel object. The submit() function updates the current buffer address and the current buffer size properties of the channel object. When the ISR runs as a result of a receive or transmit interrupt from the McBSP, it reads the channel object and uses properties such as the current buffer address and size to do the appropriate data exchange to or from the McBSP. The ISR also decrements the buffer size. When the buffer size reaches zero, the ISR has finished processing the current buffer and notifies the adapter by invoking the appropriate callback. An example of this type of controller is the C5402 McBSP controller that is included with this application note.

6.1.1 The Channel Object

The channel object shares information between the submit() function and the McBSP ISR. Here is an example of the channel object used in the C5402 McBSP controller:

```
typedef struct ChanObj {
    Bool        inuse;          /* TRUE => channel has been opened */
    LIO_Mode    mode;           /* LIO_INPUT or LIO_OUTPUT */

    Uns         *bufptr;        /* pointer *within* current buffer */
    Uns         bufcnt;         /* remaining samples to be handled */
    Uns         bufsize;        /* size of this buffer */

    LIO_Tcallback callback;     /* used to notify client when I/O complete */
    Arg         callbackArg;
} ChanObj, *ChanHandle;
```

- The **inuse** variable is set to true by the open() function and should be set to false by the close() function.
- The **mode** determines the direction of the channel and what hardware is allocated for the controller.
- The **bufptr** and **bufsize** are used to communicate with the ISR.
- The **bufcnt** is used to communicate with the adapter. The adapter is capable of handling multiple submit() calls to pass several buffers to the controller. Bufcnt is used to keep track of the amount of data that has been submitted to the controller.

In this example, the controller accepts only one buffer of data at a time. So if bufcnt is not equal to zero at the beginning of submit(), submit() ends and returns a failure to the adapter. The adapter uses this information to stop calling submit(). The last two members of the channel object are the callback and callback argument. These two channel properties are used by the ISR to synchronize with the adapter.

This device controller statically declares an array of channel objects with an element for each channel that it supports. This controller is used to send and receive buffers of samples to and from the AD50 codec located on the C5402 DSK. The codec is connected to the DSK through one of the DSP's McBSPs. These McBSPs are full-duplex; they have both receive and transmit channels. Thus, the controller also supports two channels and has two channel objects. These objects are initialized globally. Each channel object's mode property is set to either LIO_INPUT or LIO_OUTPUT.

6.1.2 The setup() Function

The DSK5402_MCBSP_AD50_setup() also needs to perform hardware setup of the McBSP and codec. This setup is needed for both the transmit and receive channels and should only be performed once. Therefore, it should be done in the setup() function, rather than the device controller's open() function. Performing this setup in the setup() function also minimizes the memory needed by the device driver. This setup could also be performed in the open() function if the system needed more flexibility. The choice of using setup() or open() is a design decision that has to be made for each device controller.

The setup() function uses TI's Chip Support Library (CSL) to configure and start the McBSP that is being used to communicate with the codec. Then it uses the McBSP to set up the codec properly for this application. Here is an example of some of the code in the setup() function:

```
if (curinit) {
    return;
}
curinit = TRUE;

/* open the McBSP */
hMcbsp = MCBSP_open(MCBSP_PORT1, MCBSP_OPEN_RESET);
MCBSP_config(hMcbsp, &mcbspCfg0);

/* start the McBSP */
MCBSP_start(hMcbsp, MCBSP_XMIT_START | MCBSP_RCV_START, 0x0);

/* setup and initialize the codec */
AD50_setParams(hMcbsp, &(setup->ad50));
```

This setup code is different for each device controller. The setup code is hardware dependent, while the rest of the controller interface is more standardized. Each controller uses the setup() function to set up its dependent hardware. Someone that wants to write a device controller can actually take this controller as good starting point. For example, the AD50_setParams() function call above can simply be changed to MYCODEC_setParams(). Of course, the McBSP configuration may change as well, but the CSL reduces the amount of effort needed to make any changes. The process of adapting the setup() function to the new hardware is the biggest step in converting a device controller from one device to another. Since this is code that has to be written anyway, the act of plugging it into the device controller interface requires minimal effort considering the benefits that it provides.

6.1.3 The open() Function

The setup() function sets up the hardware for the entire controller and is only called once. The device controller's open() function is called to create a channel instance of the controller. The example controller supports two channels and open() is called for each channel. The open() function enables a particular channel by changing the state of the channel's object. The open() function sets the channel object's inuse property to true. If a channel is already in use, then it cannot be opened again. Here is the piece of code from the example that checks this condition:

```
    if (ATM_setu((Uns *)&chan->inuse, TRUE)) {
        return (NULL);          /* ERROR! channel is already open! */
    }
```

Notice that a DSP/BIOS API call, ATM_setu(), is used to set the condition only if it is not already set. If it is currently set, the open() call returns a null pointer to the calling function in the adapter. The ATM call disables interrupts while the inuse variable is being modified. This ensures that two threads of different priority do not accidentally open the same channel twice.

Recall that the channel object holds information for the device controller. One of the channel object's parameters is the callback function. The callback function called by the controller when it completes a buffer. This function actually resides in the adapter, and a pointer to it is passed to the open() function. An optional argument to this function is also passed to open(). The open() function copies these values in the channel object, as the example does here:

```
    chan->callback = cb;
    chan->callbackArg = cbArg;
```

The open() function should also perform any hardware initialization that is channel-specific and not done by the setup() function. The example controller has a channel for receive and transmit sides of the McBSP. Each of these channels uses a separate ISR and hardware interrupt. The example controller enables the interrupts individually as shown in the following code:

```
if (mode == LIO_INPUT) {
        IRQ_enable(IRQ_EVT_RINT1);
    }
    else {
        IRQ_enable(IRQ_EVT_XINT1);
    }
```

6.1.4 The submit() Function

Once a channel has been opened successfully, it can be used by the application to send or receive buffers of data. When the adapter receives a new buffer from the application, it calls the controller's submit() function. This function receives three arguments: a pointer to the channel object, a pointer to the new buffer, and the size of the buffer in MAUs. The first thing that the submit() function does in the example is to make sure that bufcnt is equal to zero. Otherwise, there should not have been a call to submit() because the controller is already processing a buffer. If bufcnt is not equal to zero, submit() returns failure. If bufcnt is equal to zero, submit() proceeds by copying the arguments into the channel object. The arguments that it modifies are the size of the current buffer, the address of the current buffer, and the buffer count. The bufcnt property is set last to properly synchronize with the ISR. When this is complete, the submit() returns successfully. Here is the code for the submit() function from the example:

```
static Int submit(Ptr chanp, Ptr bufp, Uns nmaus)
{
    ChanHandle chan = (ChanHandle)chanp;

    if (chan->bufcnt != 0) {
        return (-1);         /* ERROR!  we only support one I/O request */
    }

    /* 'bufsize' is only used as parameter for callback function */
    chan->bufsize = nmaus;

    chan->bufptr = (Uns *)bufp;

    /* 'bufcnt' must be set last to synchronize with ISR */
    chan->bufcnt = nmaus;

    return (0);              /* success */
}
```

6.1.5 The ISRs

The ISR runs as a response to a receive or transmit interrupt from the McBSP. For the McBSP controller, there are separate ISRs for each event. If bufcnt is equal to zero when the ISR runs, this is an error condition and the controller should handle this case. In the example, the ISRs either do a dummy read for the receive ISR or a dummy write for the transmit ISR. If bufcnt is not equal to zero, the ISR reads or writes a sample to or from the buffer address in the channel object. Next, the ISR increments the buffer pointer and decrements the bufcnt variable. When bufcnt reaches zero, the ISR has either filled or emptied a buffer. It now needs to notify the adapter that a buffer is complete. The ISR could simply make a function call into the adapter to notify it that a buffer is complete; however, this requires the controller to know specific information about the adapter and it makes the controller adapter-specific. To avoid this, the controller interface uses a callback type of signaling. The channel object contains a pointer to a function that is initialized by the adapter. This function pointer is used by the ISR to call the function specified by the adapter. There are two arguments to the callback, the buffer size and an optional callback argument that is also specified in the channel object. Here is the code for the receive interrupt for the McBSP.

```
Void DSK5402_MCBSP_AD50_rxisr(Void)
{
    ChanHandle  chan = &chans[LIO_INPUT];

    if (chan->bufcnt == 0) {   /* error -- spurious interrupt or missed real-time */
        MCBSP_read(hMcbsp);           /* toss data */
        return;                         /* ERROR */
    }

    *chan->bufptr = MCBSP_read16(hMcbsp);

    chan->bufptr++;
    chan->bufcnt--;

    /* Is this buffer finished? */
    if (chan->bufcnt == 0) {
        (*chan->callback)(chan->callbackArg, chan->bufsize);
    }
}
```

6.1.6 Summary

The device controller for the McBSP that we have been discussing is a good example of any sample-by-sample controller. These controllers use the channel object to keep the state of the channel and share that state with the ISR. The ISR runs for every sample that the controller reads or writes, and update the information in the channel object appropriately. These types of controllers map well to any type of hardware that deals directly with individual samples.

6.2 Buffering Enabled by the EDMA on the C6x11 DSK

A sample-by-sample device controller involves a context switch to and from the ISR for every sample that it processes. Using a Direct Memory Access (DMA) unit to perform the sample transfers can minimize this overhead. When using the DMA, the ISR only runs once per buffer as opposed to every sample. A controller can use the DMA to minimize its overhead and overall impact on the application. A DMA enabled controller uses the DMA and an underlying I/O device to send and receive buffers of data. The initialization routines of this controller set up the DMA, the underlying device, and any logic necessary to tie these two devices together. When the controller receives a buffer from the adapter, it submits the buffer as a job to the DMA. The DMA then transfers the samples between the buffer and the underlying hardware. Any sample-by-sample synchronization that is needed is handled by the hardware setup. When the DMA finishes with a buffer, an ISR runs to signal the adapter that a buffer has been completed. A good example of this type of controller is the C6x11 EDMA enabled McBSP device controller that comes with this application note.

The overall flow of data through the C6x11 EDMA enabled device controller is very similar to the sample-by-sample device controller discussed above. The hardware, including the EDMA, is initialized by the setup() function. The adapter calls the open() function to create a new channel, which now includes an EDMA channel. The open() function initializes the appropriate channel object and marks it as in use. When the adapter receives a new buffer from the application, it calls the controller's submit() function. The submit() function takes the information passed to it by the adapter and updates the channel object. It then uses this information to program an EDMA transfer synchronized to the McBSP. When the McBSP needs a new sample to send or receive, it notifies the EDMA through hardware. The EDMA ISR runs when the EDMA has finished transferring a buffer. It then calls the callback function referenced by the channel object.

The EDMA can be reprogrammed at the next call to submit(), to set up the next buffer transfer. This method would follow very closely with the sample-by-sample controller that was discussed earlier, but it would not take full advantage of the EDMA's capabilities. Some DMAs, including the EDMA, have the capability to "auto-reload" themselves with a new transfer at the completion of a transfer. Since the EDMA reloads itself, the CPU is liberated from this task. The parameters for the next transfer can either be stored in a group of global reload registers, a dedicated set of registers for each channel, or a piece of memory called a parameter RAM (PRAM). The EDMA uses the last option: it has an area of memory that is dedicated to reloading EDMA transfers. For more information on the EDMA and its auto-reload capability, see the TMS320C6000 Peripherals Guide (SPRU190).

The EDMA uses the concept of a link pointer to manage auto-reloads. The EDMA channel is set up to link to a new transfer and an address to one of the locations in the PRAM is used for the reload address. The locations in PRAM can also have links to reload locations (i.e. other locations in the PRAM) so that a linked list of transfers can be set up in the EDMA. This capability allows the EDMA to manage multiple buffers. The example controller takes advantage of this ability by allowing the application to submit multiple buffers to it. Recall that in the sample-by-sample case, the application could only submit one buffer at a time. The controller uses the channel object to store the number of buffers and the pointers to those buffers.

6.2.1 The Channel Object

The channel object shares information between the submit() function and the EDMA ISR. Here is an example of the channel object used in the C6x11 EDMA controller:

```
/* Driver channel object structure */
typedef struct ChanObj {
    Uns           inUse;                /* True if channel has been opened */
    LIO_Mode      mode;                 /* LIO_INPUT or LIO_OUTPUT */
    Int           submitCount;          /* number of submit calls pending */
    EDMA_Handle   xferPram;             /* handle to transfer PaRAM */
    EDMA_Handle   pramTbl[MAXSUBMITCNT]; /* handles to link PaRAMs */
    EDMA_Handle   prevPramPtr;          /* points to link PaRAM last used */
    Int           writeIndex;           /* indice of next PaRAM to write to */
    Int           readIndex;            /* indice of next PaRAM to read from */
    Int           tcc;                  /* channel transfer complete code */
    LIO_Tcallback callback;             /* called when I/O complete */
    Arg           callbackArg;          /* argument to callback function */
} ChanObj, *ChanHandle;
```

- The **submitCount** member of the channel object stores the number of buffers that have been submitted to the controller.
- The **xferPram** member points to the location that contains the registers that program a real EDMA transfer.
- The **pramTbl[ ]** member stores pointers to the PRAM locations that are used to reload the transfer.
- The **prevPramPtr** points to the last PRAM area that was set up by a submit() job.
- The **readIndex** and **writeIndex** point to the next PRAM area to be read or written. The writeIndex and the readIndex are used as indices to the pramTbl[] array and point to the next array member to be set up or processed.
- The **tcc** is the channel transfer complete code allocated in open().

These variables are used by the controller to maintain the state of the linked list of transfers that have been submitted by the LIO adapter.

6.2.2 The setup() Function

The setup() function is used to setup hardware used by all channels of the device controller. A structure is passed to setup() from the application that enables or disables cache coherency calls in the controller. If the cache is enabled and the buffers are off chip then either the application or the device driver must handle the cache coherency calls. The default setting is for the controller to handle these calls. The EDMA controller initializes the McBSP and the codec inside the setup() function. It also enables the EDMA interrupt inside of setup(). To make sure that these actions are only performed once, the setup() function checks if it has already been called.

6.2.3 The open() Function

The open() function sets up a particular channel to the EDMA controller. This includes the setup of the appropriate EDMA channel (either receive or transmit). The function starts out by creating a configuration structure for the EDMA channels that are used by the controller. It then checks to make sure that the channel that is being opened is not already being used by something else by checking the inUse member of the channel object. It sets the callback pointer and the callback argument for the channel object with the parameters that were passed to it. The open() function then allocates a transfer complete code(TCC) for the EDMA channel that the controller is using. The TCC tells the EDMA which channel causes the EDMA interrupt. This is important since all of the channels use a single EDMA interrupt.

The open() function allocates the PRAM used by the controller. It allocates a PRAM location for each transfer that it can manage. The controller specifies this value as a constant named MAXSUBMITCOUNT. This value allows the controller author to specify the number of pending transfers that it can maintain for a given channel. This number may be limited by the architecture of the controller itself, or the underlying hardware that it uses.

The open() function then modifies the generic configuration that it created to make it channel-specific. If the channel is being opened for input, the source of the transfer is set to the serial ports receive register, the source index is set to none since the receive register is a fixed address, and the destination index is set to increment. If the channel is being opened for output, the destination of the transfer is set to the transmit register in the serial port, the destination index is set to none, and the source index is set to increment. When these changes have been made, the configuration is written to the EDMA registers that control the channel that is being opened, either input/receive or output/transmit. These values are also written to all of the PRAM locations that were allocated for the given channel. Here is the code that does channel dependent configuration and writing of the registers using CSL.

```
    if (chan->mode == LIO_INPUT) {
        /* allocate the EDMA transfer PaRAM for the channel */
        chan->xferPram = EDMA_open(EDMA_CHA_REVT0, 0);

        /* program the receive-specific parameters into the config structure */
        cfgEdma.src = MCBSP_getRcvAddr(hMcbsp);
        cfgEdma.opt |= EDMA_FMK(OPT, PRI, EDMA_OPT_PRI_HIGH);
        cfgEdma.opt |= EDMA_FMK(OPT, SUM, EDMA_OPT_SUM_NONE);
        cfgEdma.opt |= EDMA_FMK(OPT, DUM, EDMA_OPT_DUM_INC);
    }
    else {
        /* allocate the EDMA transfer PaRAM for the channel */
        chan->xferPram = EDMA_open(EDMA_CHA_XEVT0, 0);

        /* program the transmit-specific parameters into the config structure */
        cfgEdma.dst = MCBSP_getXmtAddr(hMcbsp);
        cfgEdma.opt |= EDMA_FMK (OPT, PRI, EDMA_OPT_PRI_LOW);
        cfgEdma.opt |= EDMA_FMK (OPT, SUM, EDMA_OPT_SUM_INC);
        cfgEdma.opt |= EDMA_FMK (OPT, DUM, EDMA_OPT_DUM_NONE);
    }

    /* program the EDMA transfer and link PaRAM with the config structure */
    EDMA_config(chan->xferPram, &cfgEdma);
    for (i=0; i < MAXSUBMITCNT; i++) {
        EDMA_config(chan->pramTbl[i], &cfgEdma);
    }
```

At this point, all the controller needs are the buffers to send or receive. These are provided by the application through the adapter by the call to the submit() function.

6.2.4 The submit() Function

The submit() function receives new buffers from the adapter. If the controller's submit count has reached the MAXSUBMITCOUNT, submit() returns a –1 to indicate failure. Otherwise, the submit() function adds the buffer to the linked list of transfers. This management is done by modifying values in the channel object. The submit() function disables the EDMA channel while it modifies these values to prevent the submitCount parameter from changing when the EDMA ISR runs. It then uses the writeIndex member of the channel object to find which PRAM location to use for the next transfer. The following piece of code sets the source or destination of this PRAM location based upon the mode of the channel (LIO_INPUT or LIO_OUTPUT).

```
    /* load the buffer pointer into the EDMA, flush cache if needed */
    if (chan->mode == LIO_INPUT) {
        EDMA_RSETH(pramPtr, DST, (Uint32)bufp);
    }
    else {
        if (cacheCalls == TRUE) {
            CACHE_flush(CACHE_L2, bufp, nmaus / sizeof(Uint16));
        }

        EDMA_RSETH(pramPtr, SRC, (Uint32)bufp);
    }
```

The submit() function sets the size of the transfer based upon the size of the buffer that it was given.

Logic in the submit() function then decides what actions need to be taken to add this to the linked list of transfers. If the submit count is equal to 0, then this is the first transfer or the EDMA exhausted the other transfers. In this case, there is no active transfer. The values that are in the PRAM need to be copied to the active transfer registers to start a new EDMA transfer. If there is an active transfer when a new one is submitted, it needs to be added to the linked list of transfers. The prevPram member of the channel object is used to find the last link in the list. This link is modified so that its link property points to the new transfer. The prevPram handle is updated to point to the new transfer. At this point, the writeIndex and submitCount members of the channel object are incremented. Finally, the EDMA channel is reenabled and the submit() functions returns with value of 0 indicating that a new buffer was successfully added.

6.2.5 The ISR

The ISR in the EDMA enabled controller is different from the sample-by-sample case. In the sample-by-sample example, there is a separate receive and transmit ISR for each channel. The EDMA only has one interrupt for all of its channels. Therefore, the ISR has to figure out which channel (receive, transmit, or both) caused the ISR to run. Other than this bit of code, the ISR is actually quite simple. Since the buffer has already been processed, all that needs to be done is decrement the submitCount, increment the read index, and call the appropriate callback with the correct arguments. Here is the code that handles the EDMA channel that transmits data.

```
    /* if interrupt (also) occured because the transmit buffer is full */
    chan = &chans[LIO_OUTPUT];
    if (chan->inUse == TRUE && EDMA_intTest(chan->tcc)) {
        if (chan->submitCount == 0) {
            LOG_error("Spurious EDMA interrupt in DSK6X11_EDMA_AD535_isr", 0);
        }

        /* Reset the pending Flag */
        EDMA_intClear(chan->tcc);

        chan->submitCount--;

        chan->readIndex = nextIndex(chan->readIndex);

        /* set up pointer to this interrupt's corresponding submit job */
        pramPtr = (EDMA_Config*)
          EDMA_getTableAddress(chan->pramTbl[chan->readIndex]);

        /* call the callback function */
        (*chan->callback)(chan->callbackArg, (pramPtr->cnt * sizeof(Uint16)));
    }
```

6.2.6 Summary

The C6711 EDMA Controller provides a good example for anyone that wants to write a DMA enabled controller. The C5402 DMA Controller is a less complex example and provides a double-buffered implementation. Both of these controller's can be found in the source code examples. Using the DMA has several advantages, but it also involves some challenges. The biggest challenge is the management of the buffers that have been submitted to the controller. The EDMA PRAM can be used as a linked list to manage these buffers. The controller can use pointers and indexes stored in the channel object to store information needed for this management. This method solves this problem well for the EDMA, but other DMA controllers may need a different solution.

6.3 Making It Your Own

These examples provide a good learning foundation for the device controller author, but the ultimate goal is to actually create new device controllers for different devices. So, how do you leverage these examples to create your own device controller? Many of the details that need to be considered are actually handled by the interface itself, but there are some issues that need to be specifically addressed.

The example drivers that we have discussed provide a good template for a new device controller. A device driver developer can take the files that make up one of the examples, copy them, and change them to accommodate the hardware. Some functions will probably need to be changed. For example, the submit() function that sets up the hardware usually must be different. The open() and close() functions may need to be changed as well. The ctrl() function is very device-specific; it probably needs to be changed so that the device controller can reveal the specific capabilities of the new hardware. Again, the code that needs to change in these functions is the hardware-specific code. The LIO-specific code in these functions can be reused if the channel object does not need to change. If the submit() function deals primarily with the channel object, it may not need to be changed at all. However, if it is used to interface with hardware as in the DMA enabled examples, the DMA code may need to be modified to support the new hardware.

The examples all use the Chip Support Library (CSL) to program the hardware. So even if a device controller author has to make significant changes to the examples for their hardware, all they have to do is modify CSL code. The CSL code is easy to read and standardized so that the author can quickly see how the hardware is being set up in the examples, and what changes might need to be made.

Every device controller has to have a different name. The LIO model uses a naming convention to avoid namespace collisions due to different drivers using the same function names. A side benefit to the naming convention allows drivers to be changed without recompiling application code since the driver functions are accessed through a function table. With this approach, only one external symbol needs to be defined for each driver. A naming convention is used for the table symbol, further simplifying the system issues. The naming convention distinguishes each function table by board, on-chip peripheral, and off-chip peripheral. For example, the source code included with this application note implements an EDMA-based driver for the TI TMS320VC6711 DSP Starter Kit' s AD535 audio codec. Therefore the device controller table name is DSK6X11_EDMA_AD535. A step in converting this device controller to a new device controller, is changing the name.

A device controller is composed of the following files:

- dsk6x11_edma_ad535.c
- dsk6x11_edma_ad535.h
- ad535.c
- ad535.h
- dsk6x11_edma_ad535.pjt

To convert this device controller, start by following these steps:

1. Create a folder called “**MYBOARD_MYPERIPHERAL_MYHARDWARE**” in the **c:\lio\src** folder. This should be the name of the new controller, as discussed above.
2. Choose Project->New. Type in **MYBOARD_MYPERIPHERAL_MYHARDWARE** for the project name in the folder you created and click Save.
3. Copy the *.c and *.h files listed above to the new folder.
4. Change the names of the files to **MYBOARD_MYPERIPHERAL_MYHARDWARE.c** and **MYBOARD_MYPERIPHERAL_MYHARDWARE.h** appropriately. The ad535.c and .h files are specific to the AD535 codec used in the example. Rename these files with something that represents the new hardware for this controller (i.e., MYHARDWARE.c).
5. Choose Project->Add Files to Project again. Select Source Files (*.c) in the Files of type box. Select the **MYBOARD_MYPERIPHERAL_MYHARDWARE.c** and **MYHARDWARE.c** files.
6. Open the source files for the new device controller and make any changes that need to be made. For example, the names of the init() function, the ISR, and the LIO_Fxns table need to be changed to **MYBOARD_MYPERIPHERAL_MYHARDWARE.** This could be done with a simple search and replace function from DSK6X11_EDMA_AD535 to **MYBOARD_MYPERIPHERAL_MYHARDWARE.**
7. With all the changes made, the device controller can now be built into a library. The library file name should match the **MYBOARD_MYPERIPHERAL_MYHARDWARE**.c file name. Go to Project-> Build Options and choose the Archiver tab. Type the Output Filename field as **MYBOARD_MYPERIPHERAL_MYHARDWARE.l62** where .l62 represents the DSP core for which the library is being built (i.e., .l64, .l54, .l55, etc.).
8. Chose Project->Rebuild All
9. Document your new device controller. Specify the "driver name", hardware resources, configuration parameters, setup parameters and memory overhead. An example of a specification's document can be found in Appendix D: Specifications for Provided Drivers, on page 47.

This process builds a device controller library that users can link into their applications. It must be paired with the appropriate adapter library to form a DSP/BIOS Device Driver.

7 Using DSP/BIOS Device Drivers: Application Examples

This section demonstrates the use of a LIO device controller with DSP/BIOS I/O to build a DSP/BIOS Device Driver. The first example shows how to construct a "sample-by-sample" device driver using the PIP Module along with software-interrupts (SWI) on the TMS320C6711 DSK. This application example simply copies data from the input to the output buffers. The second example demonstrates a "DMA" device driver using the SIO/DEV I/O Module along with tasks (TSK) on the TMS320C5402 DSK. Again, example code copies buffers from input to output. These application examples can be used with any LIO device controller supporting both the PIP and SIO/DEV frame based I/O managers.

All of the examples code, including the PIP and SIO adapter code, is provided on the Texas Instruments web site. These examples were built with Code Composer Studio 2.1. Code excerpts and summaries are presented here for discussion.

7.1 PIP with SWIs

To use an LIO device controller with the PIP module, we must create PIP object with a notifyWriter function that fills the PIP from an LIO input channel, and a notifyReader function that drains a PIP into an LIO output channel. These are the functions of the “PIP adapter” that connects the PIP API to the LIO API.

7.1.1 Configuration Setup

DSP/BIOS objects are pre-configured and bound into an executable program image. This is done through the DSP/BIOS Configuration Tool. When you save a configuration file, the Configuration Tool creates assembly and header files and a linker command file to match your settings. These files are then linked with your code when you build your application. See the sections “Using the Configuration Tool” in the DSP/BIOS User’s Guide and/or “Creating a Configuration File” in the Code Composer Studio Tutorial for more information.

The following DSP/BIOS objects are configured in this example:

- A software interrupt, **swiEcho**, to run the **echo** function.
- Two data pipes **pipRx** and **pipTx**, to exchange data between **echo** and the PLIO Adapter.
- Configure the corresponding ISRs for the MCBSP Serial Port 0 using the HWI – Hardware Interrupt Service Routine Manager

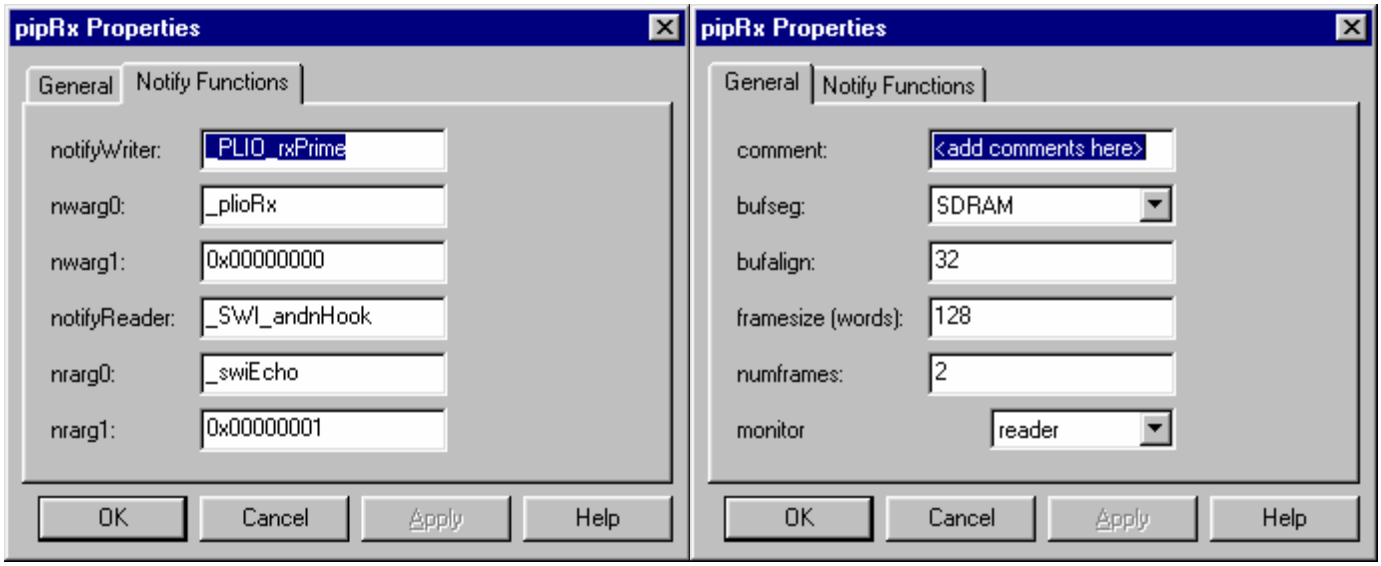
The following steps explain how to open the project with Code Composer Studio and examine the source code files and libraries used in that project.

1. If you installed Code Composer Studio in **c:\ti**, create a folder in the **c:\ti\myprojects folder**. (If you installed elsewhere, create a folder within the myprojects folder in the location where you installed.)
2. Copy all example files from the **lio.zip** file to this new folder.
3. From the Windows Start menu, choose Programs->Texas Instruments->Code Composer Studio 2 C6000 ->Code Composer Studio.
4. Choose Project->New. Type **piptest** as the project name in your new folder and click Save.
5. Choose File->New->DSP/BIOS Configuration.
6. Select the template for your DSP board and click OK.
7. Right-click on the LOG – Event Log Manager and choose Insert LOG from the pop-up menu. This creates a LOG object called LOG0.
8. Right-click on the name of the LOG0 object and choose Rename from the pop-up menu. Change the object’s name to **trace** and change the buffer length property to **256**.
9. Right-click on the SWI - Software Interrupt Manager and choose Insert SWI. Rename the new SWI0 object **swiEcho**.

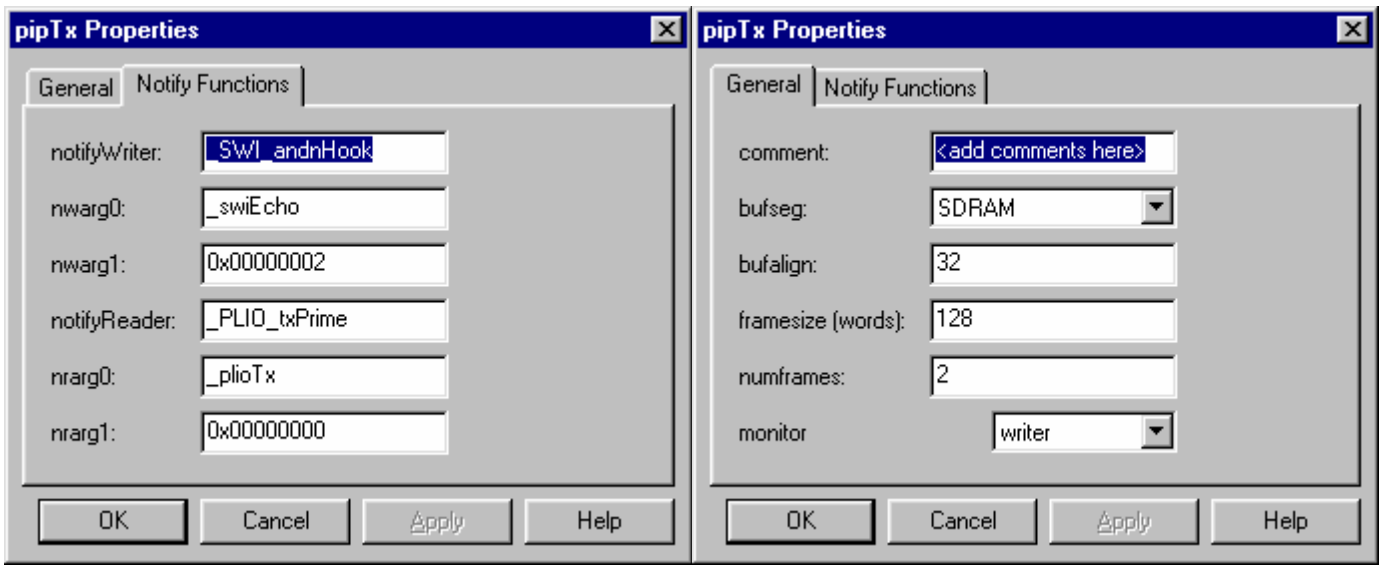
10. Right-click on **swiEcho** and select Properties from the menu. In the **swiEcho** properties window, enter **_echo** for the **function** and **3** for the **mailbox**. Click ***OK*** to save your changes.

11. Right-click on the PIP - Buffered Pipe Manager and choose Insert PIP twice. Rename the first pipe to **pipRx** and the second pipe to **pipTx**.

12. Right-click on the **pipRx** and select Properties from the menu. Enter the following properties for **pipRx**: Then click ***OK*** to save your changes.



13. Right-click on the **pipTx** and select Properties from the menu. Enter the following properties for **pipTx**:

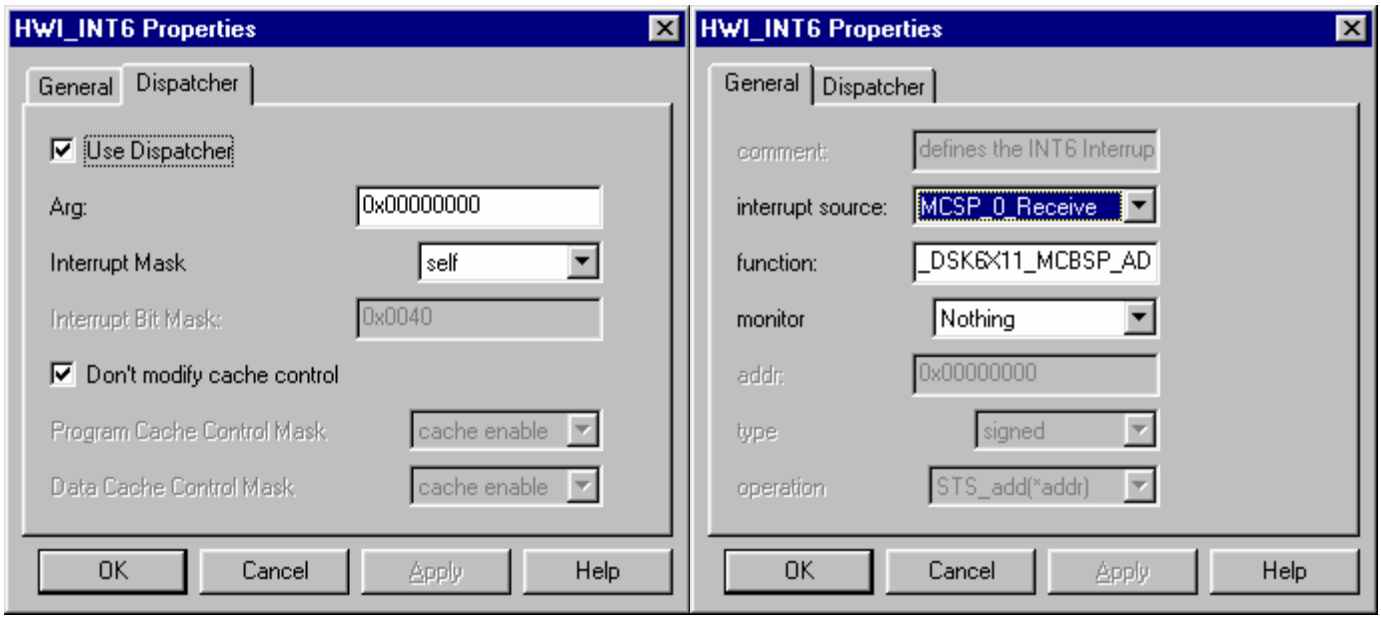


Now we need to plug in the corresponding ISRs for the serial port.

14. Click on the + next to the HWI - Hardware Interrupt Service Routine Manager to display its objects. Each of these objects corresponds to an interrupt in the interrupt vector table.

    NOTE: The interrupt locations depend upon your device controller type (either “Sample-by-sample with the MCBSP” or “Buffering enabled by the DMA”) and your TMS320 DSP (either the C5402 or the C6211/C6711). HWI_INT6 is the interrupt location for the TMS320C6211 Multichannel Buffered Serial Port 0 Receive Interrupt.

15. Right-click on the **HWI_INT6** and select properties. Choose the interrupt source that corresponds to the Multichannel Buffered Serial Port 0 Receive Interrupt (MCSP_0_Receive). Change the function to **_DSK6X11_MCBSP_AD535_rxisr**. Also, check the “Use Dispatcher” box on the Dispatcher tab similar to the following. Then click ***OK*** to save your changes.



Entering **_DSK6X11_MCBSP_AD535_rxisr** in the function field causes DSP/BIOS to set the TMS320C6211 interrupt vector table to jump to **_DSK6X11_MCBSP_AD535_rxisr** to handle the MCBSP Serial Port 0 Receive interrupt.

16. Repeat Step 15 for the **HWI_INT7** for the TMS320C6211 Multichannel Buffered Serial Port 0 Transmit Interrupt. Select its properties and choose the interrupt source that corresponds to the Multichannel Buffered Serial Port 0 Transmit Interrupt (MCSP_0_Transmit). Change the function to **_DSK6X11_MCBSP_AD535_txisr**. Also, check the “Use Dispatcher” box on the Dispatcher tab.

17. Save the configuration file as **piptest_mcbsp.cdb**. If you are asked to replace the existing file, click ***Yes***.

18. Choose Project->Add Files to Project. Select Configuration File (*.cdb) in the Files of type box. Select the **piptest_mcbsp.cdb** file and click Open. Notice that the Project View now contains piptest_mcbsp.cdb in a folder called DSP/BIOS Config. In addition, the piptest_mcbspcfg.s62 file and the piptest_mcbspcfg_c.c file are now listed as Generated Files. The following section describes how to link in the appropriate libraries and bind the controller with the application.

7.1.2 Link, Compile and Build

The linker command file binds the LIO device controller functions and controller initialization functions with the application. The CONTROLLER_FXN_TABLE, CONTROLLER_init and CONTROLLER_setup functions are shown in the piptest_mcbsp.cmd linker command file below:

```
/* Bind low level driver */
/* First include DSP/BIOS generated cmd file */

-l plio.l62
-l dsk6x11_mcbsp_ad535.l62
-l piptest_mcbspcfg.cmd

_CONTROLLER_FXN_TABLE=_DSK6X11_MCBSP_AD535_ILIO;
_CONTROLLER_init=_DSK6X11_MCBSP_AD535_init;
_CONTROLLER_setup=_DSK6X11_MCBSP_AD535_setup;
```

1. Choose Project->Add Files to Project again. Select Linker Command File (*.cmd) in the Files of type box. Select the **piptest_mcbsp.cmd** file and click Open.

2. To select the corresponding LIO Device Controller type, choose Project->Add Files to Project again. Select Object and Library Files (*.o*,*l*) in the Files of type box. Select the **dsk6x11_mcbsp_ad535.l62** file from the **\lio\lib** folder and click Open.

   WARNING:
   The device controller library depends upon your device controller type (either "Sample-by-sample with the MCBSP" or "Buffering enabled by the DMA") and your TMS320 DSP (either the C5402 or the C6211/C6711).

3. To select the PLIO adapter, choose Project->Add Files to Project again. Select Object and Library Files (*.o*,*l*). Select the **plio.l62** file from the **\lio\lib** directory and click Open.

4. Choose Project->Add Files to Project again. Select Source Files (*.c) in the Files of type box. Select the **piptest.c** file from the **\lio\apps\liotest\piptest** directory and click Open.

5. Examine the source code and demonstrate how to bind the corresponding device controller initialization with the CONTROLLER_init() and CONTROLLER_setup() functions.

```
/* ======== main ======== *
 *  Application startup function called by DSP/BIOS. Initialize the
 *  PLIO adapter then return back into DSP/BIOS.
 */
main()
{
    CONTROLLER_init();

    CONTROLLER_setup(NULL);

    /* Initialize PLIO and then bind the PIPs to LIO channels */
    PLIO_init();
    PLIO_new(&plioRx, &pipRx, LIO_INPUT, controller, NULL);
    PLIO_new(&plioTx, &pipTx, LIO_OUTPUT, controller, NULL);
```

   PLIO_init() initializes the PLIO module, while PLIO_new() initializes a specific instance in the PIP adapter for the input channel and the output channel.

6. Chose Project->Rebuild All

7.2 SIO with TSKs

In this example we implement a DEV that can be used with any LIO. The DEV functions map almost one-to-one with LIO functions. Each DEV function performs some manipulation on the SIO data structures and passes that data either to or from an LIO function. The “SIO adapter” connects the SIO/DEV API to the LIO API.

7.2.1 Configuration Setup

DSP/BIOS objects are pre-configured as well as dynamically created in this example. This is done statically through the DSP/BIOS Configuration Tool. Some DSP/BIOS objects such as SIO streams can be created dynamically in your code.

The following DSP/BIOS objects are statically configured in this example:

- A task, **tskEcho**, to run the **echo** function.
- Configure the corresponding ISRs for the DMA Channels 4 and 5 using the HWI – Hardware Interrupt Service Routine Manager
- Add the **codec** DEV object using a User Defined Device for the DSK5402_DMA_AD50 device controller

The following DSP/BIOS objects are dynamically created in this example:

- Two SIO streams, **inStream** and **outStream**, to exchange data between **echo** and the DLIO Adapter.

The following steps explain how to open the project with Code Composer Studio and examine the source code files and libraries used in that project:

1. If you installed Code Composer Studio in **c:\ti**, create a folder called audio in the **c:\ti\myprojects folder**. (If you installed elsewhere, create a folder within the myprojects folder in the location where you installed.)
2. Copy all example files from the **lio.zip** file to this new folder.
3. From the Windows Start menu, choose Programs->Texas Instruments->Code Composer Studio 2 C5000’ ->Code Composer Studio.
4. Choose Project->New. Type **siotest** as the project name in the folder you created and click Save.
5. Choose File->New->DSP/BIOS Configuration.
6. Select the template for your DSP board and click OK.
7. Right-click on the LOG – Event Log Manager and choose Insert LOG from the pop-up menu. This creates a LOG object called LOG0.
8. Right-click on the name of the LOG0 object and choose Rename from the pop-up menu. Change the object’s name to **trace** and change the buffer length property to **256**.
9. Right-click on the TSK – Task Manager and choose Insert TSK. Rename the new TSK0 object **tskEcho**.

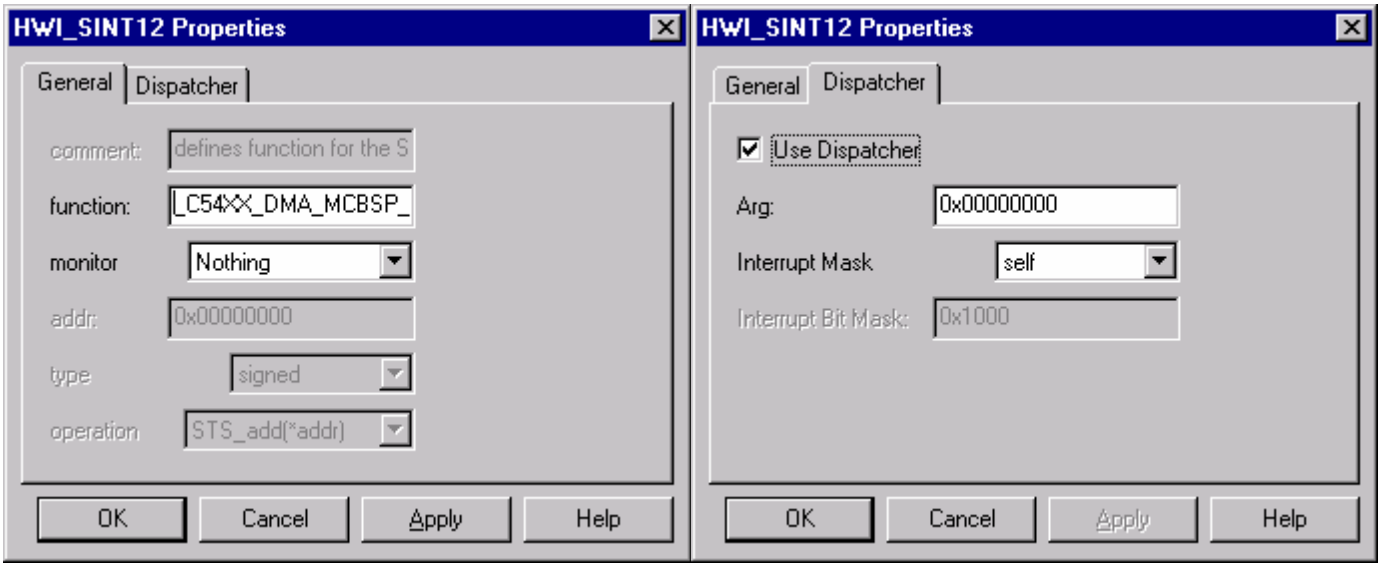
10. Right-click on **tskEcho** and select Properties from the menu. In the **tskEcho** properties window, enter **_echo** for the **function**

Next we need to plug in the corresponding ISRs for the DMA.

11. Click on the + next to the HWI - Hardware Interrupt Service Routine Manager to display its objects. Each of these objects corresponds to an interrupt in the interrupt vector table.

    NOTE: The interrupt locations depend upon your device controller type (either “Sample-by-sample with the MCBSP” or “Buffering enabled by the DMA”) and your TMS320 DSP (either the C5402 or the C6211/C6711).

    This is the interrupt location for the TMS320C5402 DMA Channel 4 Interrupt. Right-click on the **HWI_SINT12** and select properties. Change the function to **_C54XX_DMA_MCBSP_isr**. Also, check the “Use Dispatcher” box on the Dispatcher tab similar to the following:



    Click ***OK*** to save your changes.

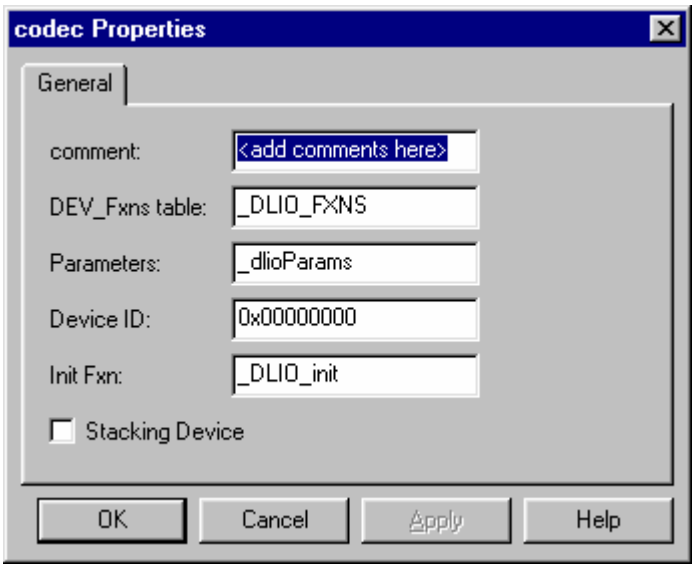
    Entering **_C54XX_DMA_MCBSP_isr** in the function field causes DSP/BIOS to set the TMS320C54x interrupt vector table to jump to **_C54XX_DMA_MCBSP_isr** to handle the DMA Channel 4 interrupt.

12. Repeat Step 11 for the **HWI_SINT13** for the TMS320C5402 DMA Channel 5 Interrupt. Select the properties and change the function to **_C54XX_DMA_MCBSP_isr**. Also, check the “Use Dispatcher” box on the Dispatcher tab and enter **0x0000001** in the Arg: field.

Next we need to plug in the corresponding DEV functions table into the User Defined Devices.

13. Click on the + next to the SIO – Stream Input and Output Manager to display its objects. Again, click on the + next for the SIO Drivers. Right-click on the User Defined Devices and choose Insert UDEV. Rename the new UDEV0 object **codec**

14. Right-click on the **codec** and select Properties from the menu. Enter the following properties. Then click ***OK*** to save your changes.



15. Save the file as **siotest_dma.cdb**. If you are asked to replace the existing file, click ***Yes***.

16. Choose Project->Add Files to Project. Select Configuration File (*.cdb) in the Files of type box. Select the **siotest_dma.cdb** file and click Open. Notice that the Project View now contains siotest_dma.cdb in a folder called DSP/BIOS Config. In addition, the siotest_dmacfg.s54 file and the siotest_dmacfg_c.c file are now listed as Generated Files. The following section describes how to link in the appropriate libraries and bind the controller with the application.

7.2.2 Link, Compile and Build

The linker command file binds the LIO device controller functions and controller initialization functions with the application. The CONTROLLER_FXN_TABLE, CONTROLLER_init and CONTROLLER_setup functions are shown in the siotest_dma.cmd linker command file below:

```
/* Bind low level driver. First include DSP/BIOS generated cmd file */

-l dlio.l54
-l dsk5402_dma_ad50.l54
-l siotest_dmacfg.cmd

_CONTROLLER_FXN_TABLE=_DSK5402_DMA_AD50_ILIO;
_CONTROLLER_init=_DSK5402_DMA_AD50_init;
_CONTROLLER_setup=_DSK5402_DMA_AD50_setup;
```

1. Choose Project->Add Files to Project. Select Linker Command File (*.cmd) in the Files of type box. Select the **siotest_dma.cmd** file and click Open.

2. To select the corresponding LIO Device Controller type, choose Project->Add Files to Project again. Select Object and Library Files (*.o*,*l*) in the Files of type box. Select the **dsk5402_dma_ad50.l54** file from the **\lio\lib** directory and click Open.

   NOTE: The controller library depends upon your device controller and your TMS320 DSP.

Next we select the corresponding LIO Adapter type, DLIO, for the SIO Module.

3. Choose Project->Add Files to Project again. Select Object and Library Files (*.o*,*l*) in the Files of type box. Select the **dlio.l54** file from the **\lio\lib** directory and click Open.

4. Choose Project->Add Files to Project again. Select Source Files (*.c) in the Files of type box. Select the **siotest.c** file from the **\lio\apps\liotest\siotest** directory and click Open.

5. Examine the source code and demonstrate how to bind the corresponding device controller initialization with the CONTROLLER_init() and CONTROLLER_setup() functions.

```
/* ======== main ======== */
main()
{
    CONTROLLER_init();
    CONTROLLER_setup(NULL);
}
```

6. Examine the source code and demonstrate how two SIO streams are dynamically created to initialize a specific instance of the adapter.

```
main()
{
    Ptr buf0, buf1, buf2, buf3;
    SIO_Attrs attrs;

    /* align the buffer to allow it to be used with L2 cache */
    attrs = SIO_ATTRS;
    attrs.align = BUFALIGN;
    attrs.model = SIO_ISSUERECLAIM;

 /* open the streams */
    inStream = SIO_create("/codec", SIO_INPUT, BUFSIZE, &attrs);
    if (inStream == NULL) {
        SYS_abort("Open of codec for input FAILED.");
    }
    outStream = SIO_create("/codec", SIO_OUTPUT, BUFSIZE, &attrs);
    if (outStream == NULL) {
        SYS_abort("Open of codec for output FAILED.");
    }
```

The SIO_create() initializes a specific instance in the SIO adapter for the input channel and the output channel.

7. Choose Project->Rebuild All

7.3 LIO Library Files

LIO device controller libraries can be found in the **\lio\lib** directory. These libraries consist of the codec source code and the device controller source code. Each controller has its own directory (i.e. **\lio\src\dsk5402ad50**) and project file.

LIO adapter libraries, PLIO and DLIO, can be found in the **\lio\src\plio** and **\lio\src\dlio** source directories. Refer to the previous sections on how to add these libraries to your application.

8 Overhead

8.1 Processing Overhead

To study the performance characteristics of the LIO driver, a series of tests are performed using the Texas Instruments DSP Starter Kits, the TMS320C6211 DSK and TMS320C5402 DSK.

Two implementations of an LIO device controller are considered. These support both PIP and SIO buffer processing and signaling through a low-level buffer management interface and a callback signaling interface. The first controller is a "sample-by-sample" interrupt driver that uses the MCBSP serial port to transfer data to or from an audio codec. The second controller uses DMA to transfer a block of data through the MCBSP to and from the audio codec. This driver receives a single interrupt per buffer, instead of an interrupt per sample as in the first case.

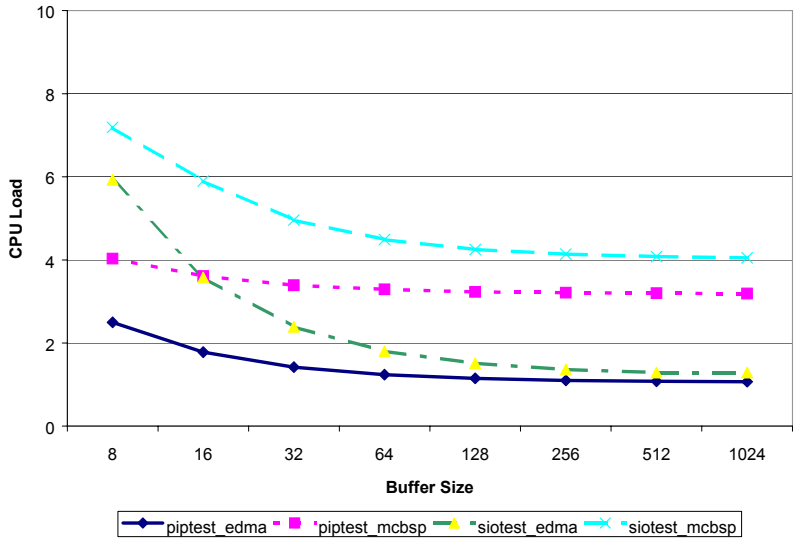


Figure 3. Buffer Size vs. CPU Load on the C6711 DSK

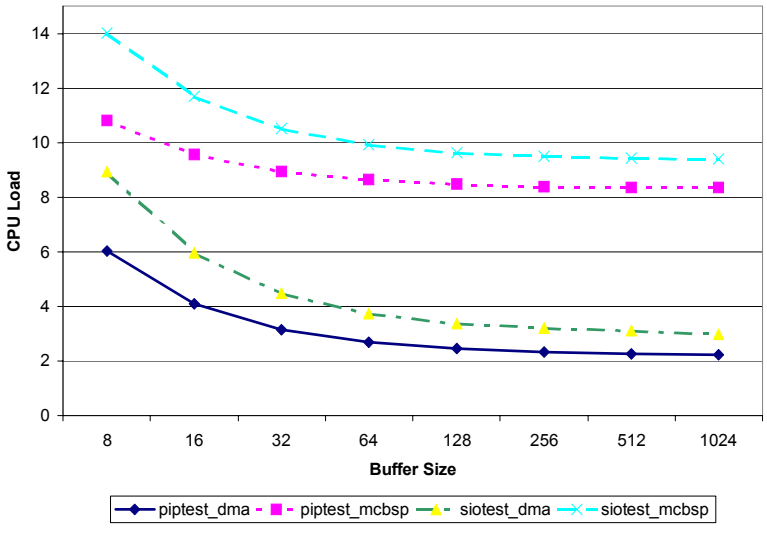


Figure 4. Buffer Size vs. CPU Load on the C5402 DSK

Figure 3 and Figure 4 show the variation of CPU load with buffer size for both the 5402 DSK and 6211 DSK. As shown, there is an extra overhead of CPU load associated with the sample-by-sample tests compared to DMA that transfers a block of data. As the buffer size increases, the CPU load approaches a constant load. This load is a criterion for selecting the optimum buffer size. In the case of SIO, it should be mentioned that increasing the buffer size might need to increase the heap size for the internal memory allocated to SIO objects in the Configuration file, otherwise data are being dropped.

8.2 Memory Overhead

This section discusses the device driver and the hardware memory overhead for a basic application footprint. This breakdown assists the system designer in planning an overall DSP solution. Table 2 and Table 3 show memory overhead for the TMS320C6711 and TMS320C5402 application examples. These tables provide details on the breakdown of the Chip Support Library (CSL) and the LIO Adapters/Device Controllers.

Table 2. Memory Overhead Example for the TMS320C6711

		piptest_mcbsp	siotest_mcbsp
Category	**Sections**	**Size decimal (8-bit bytes)**	**Size decimal (8-bit bytes)**
CSL	**.text**	3392	3392
	.cinit	504	504
PLIO/DLIO + Driver	**.text**	4032	4960
	.cinit	118	138
	.const	33	5
	.bss	138	158

Table 3. Memory Overhead Example for the TMS320C5402

		piptest_mcbsp	siotest_mcbsp
Category	**Sections**	**Size decimal (16-bit words)**	**Size decimal (16-bit words)**
CSL	**.text**	1047	1047
	.cinit	0	0
PLIO/DLIO + Driver	**.text**	799	1045
	.cinit	53	58
	.const	33	5
	.bss	43	48

8.3 Callback Overhead

To help designers better analyze their system performance characteristics, a series of timing benchmarks are performed for both PLIO and DLIO adapters. When the ISR runs as a result of a receive or transport interrupt, the device controller calls the appropriate callback function through the ISR function when the data is ready to be transferred. On the adapter side, the transfer function (called by the callback function) passes the next buffer pointer to be processed by the controller via the submit function. The elapsed time from a call to the callback function by the ISR, until to submit a buffer of data by the adapter to the controller (CPU cycles from point 1 to 2), is called: “callback overhead”. These overheads or latencies are compared for both LIO Adapters (PLIO and DLIO) running on the TMS320C6211 and TMS320C5402 DSPs and the results are shown in Table 4.

Table 4. Callback Overhead

	TMS320C5402 (CPU cycles)	TMS320C6211 (CPU cycles)
PLIO(receive)	**380**	**438**
PLIO(transmit)	**392**	**607**
DLIO	**170**	**122**

As shown in Table 4, this latency is much higher for the PLIO adapter than DLIO. The higher latency for the PLIO adapter is due to pipe (PIP) operations in the callback function that post a software interrupt. In contrast, the DLIO adapter's callback function calls the transfer function first and then the queue and semaphore operations; this results in lower overhead.

9 Conclusion

This application note has defined a low-level frame-based streaming I/O driver interface. This interface supports a wide range of application buffer management and signaling systems. Specific implementations have been shown using DMA and without DMA, and on two different Texas Instruments DSP Starter Kits. Different applications that use the same drivers have been shown with adapters for use with DSP/BIOS PIP and SIO I/O. The source code for these examples can be used as a basis for your own drivers or applications using DSP/BIOS.

10 References

For additional information, see the following sources:

Product Documentation

- *TMS320 DSP/BIOS User's Guide* (SPRU423)
- *TMS320C5000 DSP/BIOS API Reference Guide* (SPRU404)
- *TMS320C6000 DSP/BIOS API Reference Guide* (SPRU403)
- *TMS320C54x Chip Support Library API Reference Guide* (SPRU420)
- *TMS320C55x Chip Support Library API Reference Guide* (SPRU433)
- *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401)

Web Resources

- http://www.dspvillage.com

Appendix A: LIO API Reference

LIO Global Functions and Data Structures

Individual LIO controllers must provide **<controller>_init()** and **<controller>_setup()** functions. **<controller>_init()** is used to initialize the controller data structures, while **<controller>_setup()** is used initialize the device with specified parameters. **<controller>_init()** and **<controller>_setup()** must be called before any I/O channels are opened.

<controller>_SETUP

Global structure defines default parameters

Data Structure

```
<controller>_Setup <controller>_SETUP;
```

Description

<controller>_SETUP contains the default parameter settings for an individual controller. This structure is used by applications to modify the default parameters of a controller via the **<controller>_setup** API.

Example

```
DSK5402_AD50_Setup setup;

/* initialize the controller */
DSK5402_DMA_AD50_init();

/* set defaults parameters */
setup = DSK5402_AD50_SETUP;

/* change sample rate to 16kHz */
setup.control4 = 0x90;
DSK5402_DMA_AD50_setup(&setup);
```

<controller>_init

Initializes the controller module

Function

```
Void <controller>_init();
```

Arguments

None

Return Value

None

Description

This function initializes the controller module. For example, this function might initialize global data structures for the I/O channels of an individual controller.

Example

```
DSK5402_DMA_AD50_init();
```

<controller>_setup

Modify global setup parameters for a controller

Function

```
Void <controller>_setup(<controller>_Setup *setup);
```

Arguments

setup /* pointer to controller-specific params */

Return Value

None

Description

This function is used to set parameters for the controller module. **setup** is a pointer to a controller-specific structure.

Constraints

<controller>_setup must be called after <**controller>_init**, but before calling the controller's **open** function. **<controller>_setup()** is typically called from the user's main() function before interrupts are enabled.

Example

```
DSK5402_AD50_Setup setup;

/* initialize the controller */
DSK5402_DMA_AD50_init();

/* set defaults parameters */
setup = DSK5402_AD50_SETUP;

/* change sample rate to 16kHz */
setup.control4 = 0x90;
DSK5402_DMA_AD50_setup(&setup);
```

LIO Function Table

cancel

Cancel all outstanding I/O jobs

Synopsis

```
Bool cancel(Ptr chanp);
```

Arguments

```
chanp              /* channel handle */
```

Return Value

```
Bool
```

Description

cancel is used to cancel any outstanding I/O jobs that were requested via **submit**. **cancel** terminates the jobs and returns immediately. No callback functions are called after cancel returns.

cancel restores the controller to the state it was in before the first call to **submit**. **chanp** is a handle for the I/O channel as returned by a successful call to **open. cancel** returns TRUE to indicate success and FALSE if there was an error.

Constraints

cancel cannot be called for channels that were opened in 'No-Callback' mode.

close

Close an I/O channel

Synopsis

```
Bool close(Ptr chanp);
```

Arguments

```
chanp              /* channel handle */
```

Return Value

Bool

Description

close closes the I/O channel. **close** returns the controller to that state as before the channel was opened. This may include disabling the interrupt and restoring the hardware to its' default state.
chanp is a handle for the I/O channel as returned by a successful call to **open. close** returns TRUE to indicate success and FALSE if there was an error.

ctrl

Control an I/O channel

Synopsis

```
Bool ctrl(Ptr chanp, Uns cmd, Ptr args);
```

Arguments

```
chanp              /* channel handle */
cmd                /* command */
args               /* pointer to arguments */
```

Return Value

Bool

Description

ctrl is used to change the state or get the state of an individual I/O channel. **chanp** is a handle for the I/O channel as returned by a successful call to **open.** The controller-specific **cmd** and **args** are used by the controller to modify or return the state of the channel. Clients of **ctrl** must include the header file that defines the controller-specific **cmd** and **args** structures. **ctrl** returns TRUE to indicate success and FALSE if there was an error.

open

Open an I/O channel

Synopsis

```
Ptr open(String name, LIO_Mode mode, Ptr args,
LIO_Tcallback cbFxn, Arg cbArg);
```

Arguments

```
name               /* optional name string */
mode               /* input or output */
args               /* channel-specific args */
cbFxn              /* callback function */
cbArg              /* callback argument */
```

Return Value

```
Ptr chanp          /* channel pointer */
```

Description

open opens an I/O channel and returns a pointer to a controller-specific channel object. This pointer is passed to **submit, cancel, ctrl** and **close. open** returns NULL to indicate that the channel open failed. The **name** parameter can be used to pass a character string to the controller. This name string is used in a controller-specific manner. **mode** must be either LIO_INPUT or LIO_OUTPUT to specify an input

channel or output channel. **args** is a pointer to a controller-specific arguments structure. This structure can be used to specify optional parameters (e.g., sample rate, BAUD rate, etc.) for the controller. **args** can be NULL if the controller does not support controller-specific arguments or to indicate default arguments. **cbFxn** and **cbArg** are used to specify the function and argument to be called when an I/O job completes. This is useful in 'callback' mode where **submit** starts the I/O and returns immediately. In callback mode, the application is notified when I/O completes by calls to the callback function.

The function prototype for the callback function is:

```
Void callback(Arg cbArg, Uns nmaus);
```

If **cbFxn** is not NULL, the callback function is called every time an I/O job is finished. **nmaus** specifies the number of MAUs that were successfully input or output. It is less than or equal to the **nmaus** that were passed to **submit.** If **cbFxn** is NULL, the controller is opened in 'no-callback' mode. In no-callback mode, the **submit** function returns the number of MAUs that were successfully input or output.

An individual controller may support callback mode, no-callback mode, or both.

Constraints

The **open** function *must* check the **cbFxn** parameter and return NULL if the particular model (callback or no-callback) is not supported.
The callback function may be called from interrupt level with one or more interrupts disabled. The callback function may also be called at application level.

submit

Submit a buffer to a channel for I/O

Synopsis

```
Int submit(Ptr chanp, Ptr buf, Uns nmaus);
```

Arguments

```
chanp           /* channel handle */
buf             /* I/O buffer */
nmaus           /* size in MAUs */
```

Return Value

```
Int             /* < 0 => error */
```

Description

submit is used to initiate an I/O operation. **chanp** is a handle for the I/O channel as returned by a successful call to **open. buf** is a pointer to the buffer to be input or output. **nmaus** specifies the number of MAUs to be input or output. In callback mode, **submit** returns '0' if the controller can handle the I/O request and a negative value (e.g., -1) if the controller cannot handle the request. The callback function is called when the I/O is complete. It specifies the number of MAUs successfully processed. In no-callback mode, **submit** returns the number of MAUs that were successfully input or output or a negative value if there was an error. The callback function is never called in no-callback mode.

Constraints

submit may be called from interrupt level code or with interrupts disabled. Submit code should therefore be optimized to minimize interrupt latency.

Appendix B: PLIO Adapter

PLIO Adapter

Interface between LIO controllers and Input/Output (PIP) pipes

Description

The PIP adapter, also referred to as PLIO, is designed to obtain a buffer from the application through the buffer manager and present it to the controller for consumption. The adapter recognizes when the controller is finished processing the buffer and send it back to the application through the buffer manager. This communication is accomplished with a minimal amount of overhead and complexity.

Configuring a PLIO Device

To use the PLIO adapter insert the following objects inside the Configuration tool:

- Add a software interrupt from the SWI - Software Interrupt Manager and choose Insert SWI. Rename the new SWI0 object **swiEcho**. In the **swiEcho** properties window, enter **_echo** for the **function** and **3** for the **mailbox**.
- Add pipe objects from the PIP - Buffered Pipe Manager and choose Insert PIP twice. Rename the first pipe to **pipRx** and the second pipe to **pipTx**. The length of the buffers should be the same and can be any size. The **pipRx** notifyWriter function setting should be **PLIO_rxPrime(plioRx)** and the notifyReader function settings should be **SWI_andn(swiEcho,1)**. The **pipTx** notifyWriter function setting should be **SWI_andn(swiEcho,2)** and notifyReader function settings should be **PLIO_txPrime(plioTx)**.

  The PLIO adapter uses the following basic types of functions:

  - **Prime functions.** The PIP buffer manager calls rxPrime and txPrime when the application sends a buffer to the device driver. These functions use DSP/BIOS API calls to obtain a buffer from the buffer manager and present it to the controller. The "prime" functions are the signaling interface between the application and the adapter.
  - **Callback functions.** The rxCallback and txCallback functions are the signaling interface between the controller and the adapter. During driver setup, the adapter tells the controller which functions to call when it finishes with the buffer. This callback signals the adapter when a buffer is ready to be sent back to the buffer manager and ultimately, the application.
  - **Transfer function.** This function calls the device controller's submit() function. The submit() function of the controller receives a buffer from the adapter and then communicates the new buffer information to the ISR. This communication is done through the channel object.

  The PLIO adapter uses these functions to communicate between the application and the controller. This is shown in Figure 5. Arrows with full or empty boxes attached indicate buffer flow; simple arrows indicate critical function calls.

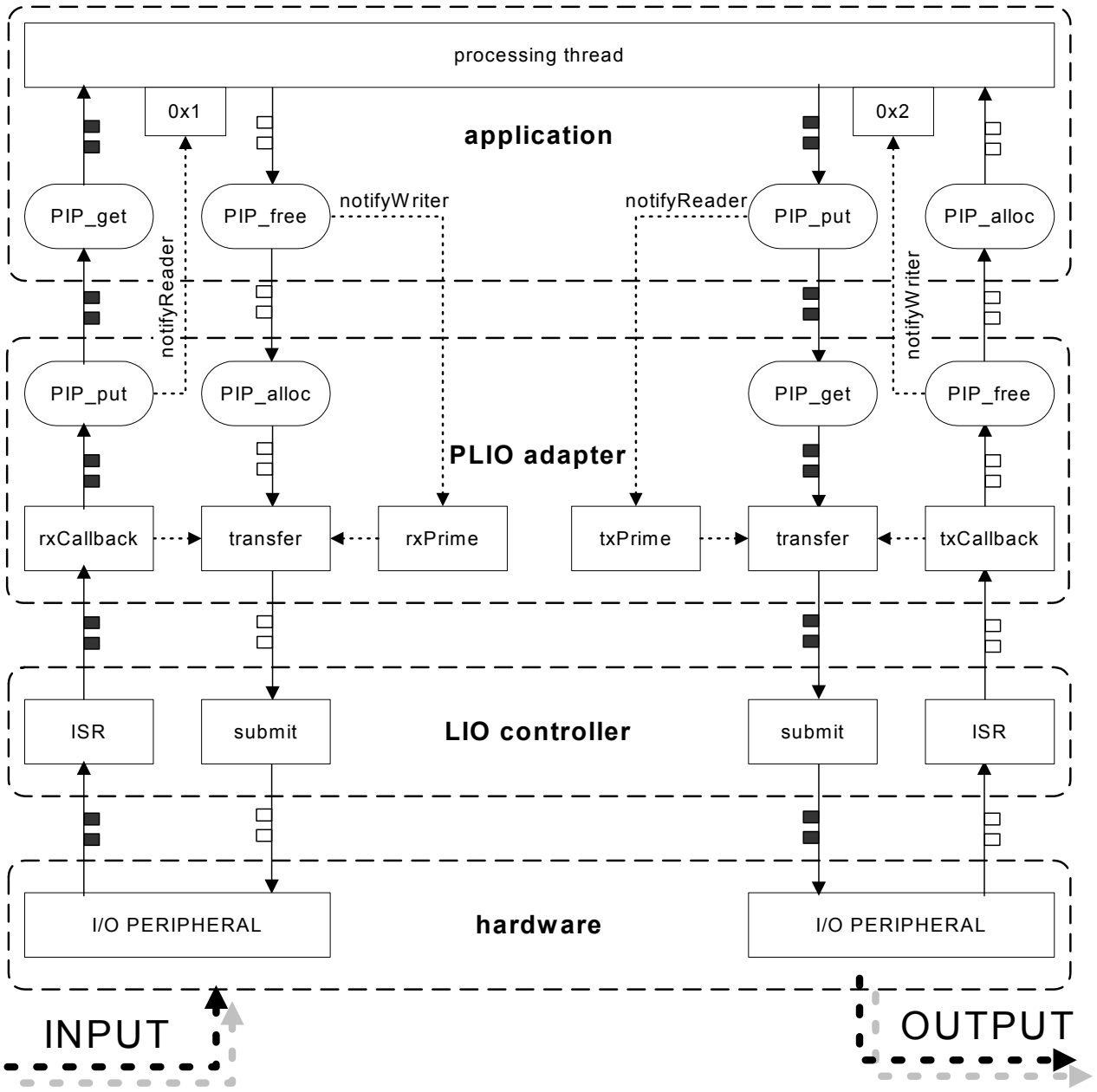


Figure 5. PLIO Adapter Buffer Flow

PLIO API Reference

PLIO_ctrl

Pass a control call to the controller

Function

```
Bool PLIO_ctrl (PLIO_Handle plio, Uns cmd, Ptr arg);
```

Arguments

```
plio    /* pointer to channel's PLIO object */
cmd     /* LIO command */
arg     /* LIO argument */
```

Return Value

Bool

Description

PLIO_ctrl is implemented as a macro and will pass its arguments directly to the LIO ctrl function.

Constraints

Specific to the LIO controller in use.

PLIO_init

Initialize the PLIO module

Function

```
Void PLIO_init();
```

Arguments

None

Return Value

None

Description

This function initializes the PLIO adapter module.

Example

```
PLIO_init();
```

PLIO_new

Initialize PLIO object for a channel and open controller

Function

```
Void PLIO_new(PLIO_Handle plio, PIP_Handle pip,
LIO_Mode mode, LIO_Fxns *fxns, PLIO_Attrs *attrs);
```

Arguments

```
plio    /* pointer to channel's PLIO object */
Pip     /* pointer to channel's PIP object */
mode    /* input or output mode */
fxns    /* pointer to LIO interface function table */
attrs   /* pointer to attributes data structure */
```

Return Value

None

Description

PLIO_new initializes the PLIO object associated with a new channel, and calls the open() function of the LIO controller for the channel. PLIO_new must be called to initialize the I/O prior to any transfers occurring. The last argument is for various attributes. If no attributes are specified the attrs parameter may be NULL. PLIO_new is typically called from main.

If the <controller>_open() fails then PLIO_new will call SYS_abort.

Example

```
/* Bind the Rx&Tx PIPs to LIO channels */
PLIO_new(&plioRx, &pipRx, LIO_INPUT, controller,
NULL);

PLIO_new(&plioTx, &pipTx, LIO_OUTPUT, controller,
NULL);
```

PLIO_rxPrime

Supply a frame to the receiver

Function

```
Void PLIO_rxPrime (PLIO_Handle plio);
```

Arguments

plio /* pointer to channel's PLIO object */

Return Value

None

Description

PLIO_rxPrime submits one frame to the LIO receiver if the receiver can accept it. The receiver's ability to accept another frame is determined by comparing the number of frames submitted with the maximum number of frames the controller can accept ("submit limit"). The submit limit is initially the number of frames in the pipe. The submit limit is updated to reflect the maximum frames the controller can accept if and when the controller fails to accept a frame. PLIO_rxPrime is typically configured as the notify writer function of the receive pipe.

Constraints

None

PLIO_rxStart

Start receive with one or more frames

Function

```
Void PLIO_rxStart (PLIO_Handle plio, Uns frameCount);
```

Arguments

plio /* pointer to channel's PLIO object */
frameCount /* number of frames to submit */

Return Value

None

Description

PLIO_rxStart submits as many frames as the controller can accept, up to the number specified by frameCount.

Constraints

PLIO_rxStart() may only be called in main() before interrupts are enabled.

Example

```
/* Prime the receive side with empty */
/* buffers to be filled. */
PLIO_rxStart(&plioRx,
PIP_getWriterNumFrames(&pipRx));
```

PLIO_txPrime

Supply a frame to the transmitter

Function

```
Void PLIO_txPrime (PLIO_Handle plio);
```

Arguments

plio /* pointer to channel's PLIO object */

Return Value

None

Description

PLIO_txPrime submits one frame to the LIO receiver if the transmitter can accept it. The transmitter's ability to accept another frame is determined by comparing the number of frames submitted with the maximum number of frames the controller can accept ("submit limit"). The submit limit is initially the number of frames in the pipe. The submit limit is updated to reflect the maximum frames the controller can accept if and when the controller fails to accept a frame. PLIO_txPrime is typically configured as the notify reader function of the transmit pipe.

Constraints

None

PLIO_txStart

Start transmit with one or more frames

Function

```
Void PLIO_txStart (PLIO_Handle plio, Uns frameCount,
Uns initialValue);
```

Arguments

plio /* pointer to channel's PLIO object */
frameCount /* number of frames to submit */
initialValue /* value used to fill frame */

Return Value

None

Description

PLIO_txStart submits as many frames as the controller can accept, up to the number specified by frameCount.

Constraints

PLIO_txStart() may only be called in main() before interrupts are enabled.

Example

```
/* Prime transmit side with buffers of silence. */
PLIO_txStart(&plioTx, PIP_getWriterNumFrames(&pipTx), 0);
```

PLIO Internal Functions

rxCallback

Notify PLIO of receive completion

Function

```
static Void rxCallback (Arg arg, Uns nmaus);
```

Arguments

```
arg      /* pointer to channel's PLIO object */
nmaus    /* the number of MAUs received */
```

Return Value

None

Description

The rxCallBack function completes the operation started by the call to the LIO controller's submit function. The pipe associated with the receive channel is updated to show the received data. Another submit operation is started to keep the flow of data uninterrupted. The rxCallBack function is typically called from the interrupt routine that services the associated LIO channel.

Constraints

Interrupts for the channel must be disabled.

txCallback

Notify PLIO of transmit completion

Function

```
static Void txCallback (Arg arg, Uns nmaus);
```

Arguments

```
arg      /* pointer to channel's PLIO object */
nmaus    /* the number of MAUs received */
```

Return Value

None

Description

The txCallBack function completes the operation started by the call to the LIO controller's submit function. The pipe associated with the transmit channel is updated so the frame can be reused. Another submit operation is started to keep the flow of data uninterrupted. The txCallBack function is typically called from the interrupt routine that services the associated LIO channel.

Constraints

Interrupts for the channel must be disabled

Appendix C: DLIO Adapter

DLIO Adapter

Interface between LIO controllers and Streaming Input/Output (SIO) streams

Description

The SIO adapter, also called DLIO, is designed to easily integrate with the existing DEV module. This communication and synchronization is accomplished with a minimal amount of overhead and complexity.

Configuring a DLIO Device

To add a DLIO adapter, right-click on the User-defined Devices in the Configuration tools, and select Insert UDEV. From the Object menu, choose Rename and type a new name for the device. Open the Properties dialog for the device you created and modify its properties as follows.

- **DEV_FXNS table.** Type _DLIO_FXNS
- **Parameters.** Type _dlioParams. This is the name of the declared params structure for DLIO. You must declare a DLIO_Params structure (as described after this list) containing the values to use for the parameters.
- **Device ID.** Type 0 (zero)
- **Init Fxn.** Type _DLIO_init
- **Stacking Device.** Remove checkmark in the box

DLIO_Params is defined in dlio.h as follows:

```
/*  ======== DLIO_Params ========*/
typedef struct DLIO_Params {
    LIO_Fxns    *fxns;
    Ptr         args;
} DLIO_Params;
```

The device parameters are:

- **LIO function.** Specifies a pointer to the device controller functions
- **arg.** Argument to pass to the controller's open function if necessary

The DLIO adapter uses the following basic types of functions:

- **Callback functions.** The callback functions are the signaling interface between the controller and the adapter. During device driver setup, the adapter tells the controller which functions to call when it finishes with the buffer. This callback signals the adapter when a buffer is ready to be sent back to the buffer manager and ultimately, the application.

- **Transfer function.** This function calls the device controller's submit() function. The submit() function of the controller receives a buffer from the adapter and then communicates the new buffer information to the ISR. This communication is done through the channel object.

The DLIO adapter uses these functions to communicate between the application and the controller. This is shown in Figure 6. Arrows with full or empty boxes attached indicate buffer flow; simple arrows indicate critical function calls.

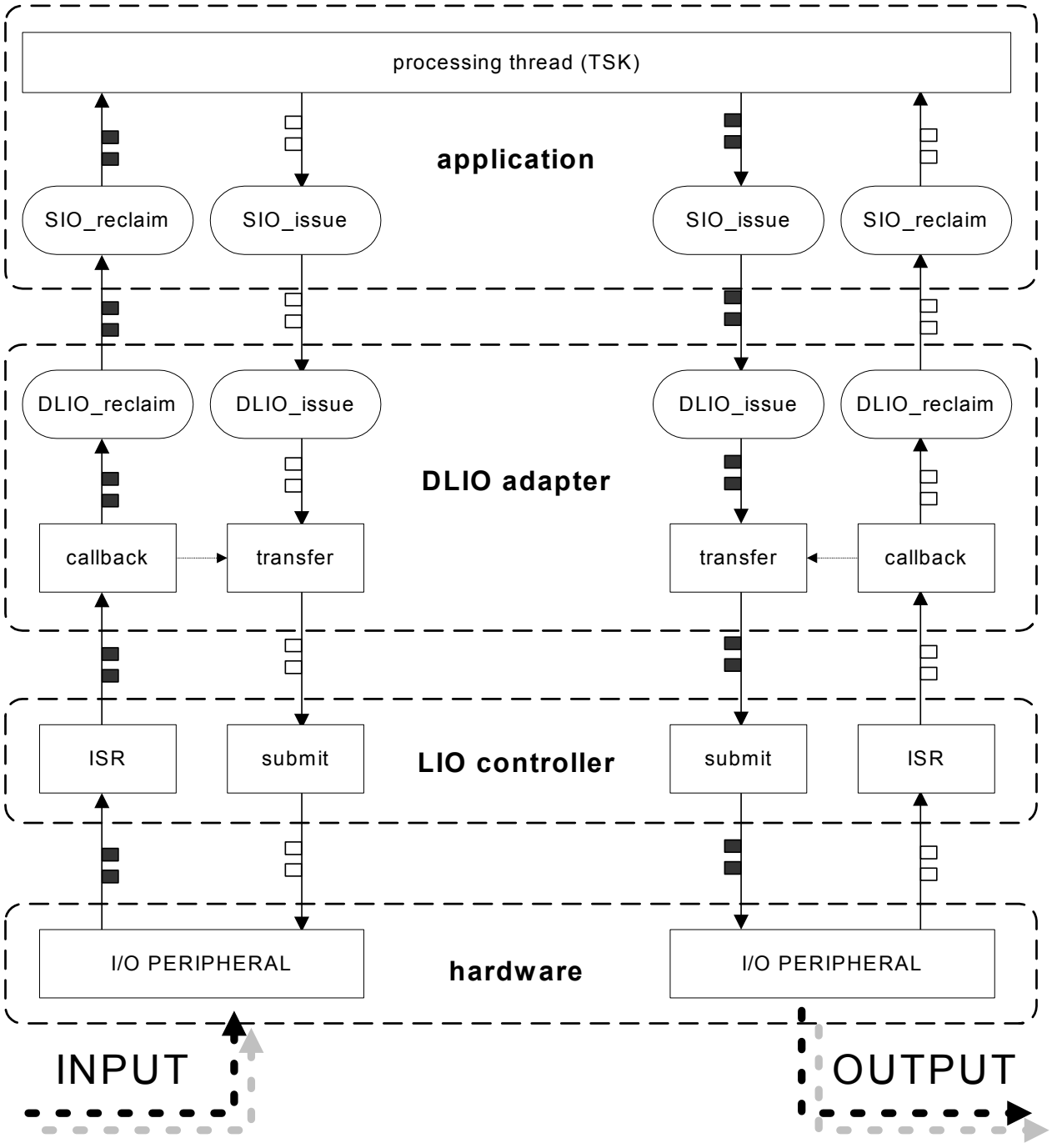


Figure 6. DLIO Adapter Buffer Flow

Data Streaming

DLIO adapters can be opened for input or output. DLIO_open allocates buffers for use by the controller. You can create streams dynamically as follows:

```
inStream = SIO_create("/codec", SIO_INPUT, 256, NULL);
outStream = SIO_create("/codec", SIO_OUTPUT,256, NULL);
```

Example

The following code example declares dlioParams as a DLIO_Params structure:

```
#include <dlio.h>

DLIO_Params dlioParams = {
   &CONTROLLER_FXN_TABLE, /* LIO function pointer */
   NULL       /* controller open argument pointer */
}
```

DLIO External Functions

DLIO_init

Initialize the DLIO module

Function

```
Void DLIO_init();
```

Arguments

None

Return Value

None

Description

This function initializes the DLIO adapter module.

Example

The UDEV properties dialog box for **Init Fxn.** Type _DLIO_init

DLIO Internal Functions

callback

Notify DLIO of data completion

Function

```
static Void callback(Arg devp, Uns nmaus);
```

Arguments

```
arg      /* pointer to streams device object */
nmaus    /* the number of MAUs received */
```

Return Value

None

Description

The callback function completes the operation started by the call to the LIO controller's submit function. The associated streams are updated to show that data has been sent/received and updated so the frame can be reused. Another submit operation is started to keep the flow of data uninterrupted. The callback function is typically called from the interrupt routine that services the associated LIO channel.

Constraints

Interrupts for the channel must be disabled.

Appendix D: Specifications for Provided Drivers

DSK5402 AD50 "Sample-By-Sample" Driver

Driver Name	Vendor	Arch	Board	Version	Doc Date	Library Name
DSK5402_MCBSP_AD50	Texas Instruments	5000	TI C5402 DSK	N/A	Feb 14 2002	dsk5402_mcbsp_ad50.l54

Hardware Interrupts Plugged

Hardware Source	Interrupt #	ISR Name	Argument
MCBSP1	10	DSK5402_MCBSP_AD50_rxisr	None
MCBSP1	11	DSK5402_MCBSP_AD50_txisr	None

Mandatory Configuration Parameters

Module	Parameter	Value	Description
HWI_SINT10	Source	MCBSP	Connect MCBSP peripheral to HWI 10
HWI_SINT10	Function	DSK5402_MCBSP_AD50_rxisr	Driver ISR
HWI_SINT10	Use Dispatcher	True	Use HWI Dispatcher
HWI_SINT11	Source	MCBSP	Connect MCBSP peripheral to HWI 11
HWI_SINT11	Function	DSK5402_MCBSP_AD50_txisr	Driver ISR
HWI_SINT11	Use Dispatcher	True	Use HWI Dispatcher

CTRL API Description

Description	Command Syntax	Argument
No CTRL commands implemented	N/A	N/A

Memory Overhead

Category	Sections	piptest_mcbsp	siotest_mcbsp
		Size decimal (16-bit words)	Size decimal (16-bit words)
CSL	.text	1047	1047
	.cinit	0	0
PLIO/DLIO + Driver	.text	799	1045
	.cinit	53	58
	.const	33	5
	.bss	43	48

Note: Reusable memory space:

piptest_mcbsp .text:init = 141 (16-bit words)
siotest_mcbsp .text:init = 51 (16-bit words)

DSK5402 AD50 "DMA" Driver

Driver Name	Vendor	Arch	Board	Version	Doc Date	Library Name
DSK5402_DMA_AD50	Texas Instruments	5000	TI C5402 DSK	N/A	Feb 14 2002	dsk5402_dma_ad50.l54

Hardware Interrupts Plugged

Hardware Source	Interrupt #	ISR Name	Argument
DMA 4	12	C54xx_DMA_MCBSP_isr	0
DMA 5	13	C54xx_DMA_MCBSP_isr	1

Mandatory Configuration Parameters

Module	Parameter	Value	Description
HWI_SINT12	Source	DMA	Connect DMA peripheral to HWI 12
HWI_SINT12	Function	C54xx_DMA_MCBSP_isr	Driver ISR
HWI_SINT12	Use Dispatcher	True	Use HWI Dispatcher
HWI_SINT13	Source	DMA	Connect DMA peripheral to HWI 13
HWI_SINT13	Function	C54xx_DMA_MCBSP_isr	Driver ISR
HWI_SINT13	Use Dispatcher	True	Use HWI Dispatcher

CTRL API Description

Description	Command Syntax	Argument
No CTRL commands implemented	N/A	N/A

Memory Overhead

Category	Sections	piptest_dma	siotest_dma
		Size decimal (16-bit words)	Size decimal (16-bit words)
CSL	.text	1413	1413
	.cinit	0	0
PLIO/DLIO + Driver	.text	962	1208
	.cinit	69	74
	.const	75	47
	.bss	62	67

Note: Reusable memory space:

piptest_dma	.text:init = 187 (16-bit words)
siotest_dma	.text:init = 97 (16-bit words)

DSK6x11 "Sample-By-Sample" Driver

Driver Name	Vendor	Arch	Board	Version	Doc Date	Library Name
DSK6X11_MCBSP_AD535	Texas Instruments	6000	TI C6x11 DSK	N/A	Feb 14 2002	dsk6x11_mcbsp_ad535.l62

Hardware Interrupts Plugged

Hardware Source	Interrupt #	ISR Name	Argument
MCBSP 0	6	DSK6X11_MCBSP_AD535_rxisr	None
MCBSP 0	7	DSK6X11_MCBSP_AD535_txisr	None

Mandatory Configuration Parameters

Module	Parameter	Value	Description
HWI_int6	Source	MCBSP	Connect MCBSP peripheral to HWI 6
HWI_int6	Function	DSK6X11_MCBSP_AD535_rxisr	Driver ISR
HWI_int6	Use Dispatcher	True	Use HWI Dispatcher
HWI_int7	Source	MCBSP	Connect MCBSP peripheral to HWI 7
HWI_int7	Function	DSK6X11_MCBSP_AD535_txisr	Driver ISR
HWI_int7	Use Dispatcher	True	Use HWI Dispatcher

CTRL API Description

Description	Command Syntax	Argument
No CTRL commands implemented	N/A	N/A

Memory Overhead

Category	Sections	piptest_mcbsp	siotest_mcbsp
		Size decimal (8-bit bytes)	Size decimal (8-bit bytes)
CSL	.text	3392	3392
	.cinit	504	504
PLIO/DLIO + Driver	.text	4032	4960
	.cinit	118	138
	.const	33	5
	.bss	138	158

Note: Reusable space from initialization:

piptest_mcbsp	.text:init = 736 (8-bit bytes)
siotest_mcbsp	.text:init = 320 (8-bit bytes)

DSK6x11 "EDMA" Driver

Driver Name	Vendor	Arch	Board	Version	Doc Date	Library Name
DSK6X11_EDMA_AD535	Texas Instruments	6000	TI C6x11 DSK	N/A	Feb 14 2002	dsk6x11_edma_ad535.l62

Hardware Interrupts Plugged

Hardware Source	Interrupt #	ISR Name	Argument
EDMA	8	DSK6X11_EDMA_AD535_isr	None

Mandatory Configuration Parameters

Module	Parameter	Value	Description
Pip Buffer	Alignment	128 Bytes	Pipe buffer alignment to match cache line size
HWI_int8	Source	EDMA	Connect EDMA peripheral to HWI 8
HWI_int8	Function	DSK6X11_EDMA_AD535_isr	Driver ISR
HWI_int8	Use Dispatcher	True	Use HWI Dispatcher

CTRL API Description

Description	Command Syntax	Argument
No CTRL commands implemented	N/A	N/A

Memory Overhead

Category	Sections	piptest_edma	siotest_edma
		Size decimal (8-bit bytes)	Size decimal (8-bit bytes)
CSL	.text	6688	6688
	.cinit	616	616
PLIO/DLIO + Driver	.text	6112	7040
	.cinit	254	274
	.const	151	123
	.bss	202	222

Note: Reusable space from initialization:

piptest_edma .text:init = 608 (8-bit bytes)
siotest_edma .text:init = 192 (8-bit bytes)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265