Writing Flexible Device Drivers for DSP/BIOS II

Jack Greenbaum
Texas Instruments, Santa Barbara

ABSTRACT

I/O device drivers play an integral role in digital signal processing (DSP) application development by separating the software components from hardware details. This promotes software reuse, modular design, and codevelopment of hardware and software. These benefits reduce time-to-market for first generation applications, and greatly reduce the level of effort for second generation designs. Many DSP applications must process streaming data, such as samples from an ADC/DAC, frame buffer/grabber, or interprocessor communication device. This application note defines a flexible application programming interface (API) for streaming I/O device drivers for use in real-time DSP applications written with the DSP/BIOS II real-time kernel. Examples are included which run on the Texas Instruments TMS320C5402 DSP Starter Kit (DSK) and the TMS320C6211 DSK. The examples demonstrate the implementation and use of the driver API with the TI eXpress DSP™ foundation software technologies DSP/BIOS II™ and Chip Support Library (CSL).

eXpress DSP and DSP/BIOS II are trademarks of Texas Instruments.

Contents

1 Low Level Device Drivers ............................................................................................................. 2
2 Constraints on a Driver API .......................................................................................................... 2
   2.1 Application Considerations ................................................................................................... 2
   2.2 Device and System Considerations .................................................................................... 3
3 LIO: A Flexible Driver API ............................................................................................................. 4
   3.1 General Guidelines, Assumptions, and Calling Conventions ............................................. 4
   3.2 Functions by Type ............................................................................................................. 5
   3.3 Detailed Descriptions ......................................................................................................... 7
4 Writings LIO drivers: Implementation Examples ........................................................................ 9
   4.1 6211 DSK: Sample-by-Sample with McBSP ................................................................... 11
   4.2 5402 DSK: DMA with Reload .......................................................................................... 14
5 Using LIO Drivers: Application Examples ................................................................................. 15
   5.1 Raw ..................................................................................................................................... 15
   5.2 PIP With SWI ................................................................................................................... 16
   5.3 SIO With TSK .................................................................................................................. 18
6 Multichannel I/O ........................................................................................................................... 19
7 Overhead ..................................................................................................................................... 20
8 Conclusion................................................................................................................................... 20

Figures

Figure 1. A Model Device with Buffers .......................................................................................... 7
1 Low Level Device Drivers

The drivers described here are intended for use in systems that require frame based streaming I/O, that is systems in which the data consists of blocks of memory to be processed as a unit with a real-time deadline. Example algorithms include vocoders and call progress tone detectors and emitters, speech recognition, mp3 players, and video processing such as frame and block based compression or real-time image analysis. The common element in all of these applications is 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 II, and the threading models with which the I/O services can be used. Then we discuss system constraints such as code and memory overhead. A flexible driver API is then presented which meets these requirements and constraints. Two example implementations are detailed, one using DMA and another using a sample-by-sample interrupt. Three ways of using these drivers are shown. The last two sections discuss multichannel drivers and overhead.

2 Constraints on a Driver API

2.1 Application Considerations

DSP/BIOS II provides two services, PIP and SIO, which support the implementation of frame-based signal processing systems. Designers may choose to implement their own buffer management. Therefore, there are at least three types of I/O strategies that the low level driver API should support. Common among the approaches is that each one manages a set of buffers to transfer to or from the peripheral and each one has a method to signal threads of program execution when a buffer transfer has completed.

Often these threads are preemptable and the thread remains blocked from executing until the required I/O has complete. DSP/BIOS II provides two types of preemptable threads, software interrupts (SWIs) and tasks (TSKs). Using preemptable threads is required for multi-rate systems such as universal port telecom, and are useful in any system where frames of different sizes or different data rates are processed. The PIP I/O objects may be used with either SWI or TSK thread types. SIO objects may only be used with TSKs. An ad hoc method could be used with either SWI or TSK.
Because DSP/BIOS II provides multiple I/O services and multiple thread types, devices drivers written for DSP/BIOS II should not constrain the user to the choice of I/O or thread type. Each approach to buffer management and signaling should be supported for a driver to be truly universal and achieve the code reuse goals. Let's look at the requirements this places on our driver API.

For example the DSP/BIOS II PIP module statically allocates I/O buffer memory at compile time. A simple allocate/free API controls which buffer is being written to or read from. When a buffer is filled the writer thread, typically a hardware interrupt, calls the PIP_put API function. In response the PIP module calls a reader notification function which schedules the data for processing. Typically the reader notify function is a DSP/BIOS II kernel call such as SWI_post (when using SWI threads) or SEM_post (when using TSK threads). When the reader completes processing the buffer it calls the PIP_free API function. In response the PIP module calls a writer notification function so that the buffer can be reused. The drivers we write for PIPs therefore must use the PIP API functions such as PIP_put to manipulate the PIP buffer management and the driver must respond to the reader and writer notify functions configured for the PIP object.

The SIO module, new in DSP/BIOS II and used with TSK threads, uses dynamically changeable buffer addresses. Typically an application allocates the SIO buffers statically or dynamically at start up and reuses the buffers cyclically. However the SIO API does not require that the buffers are used in the same order or that the buffers are ever reused. So to support the SIO API the low-level drivers must accept any buffer address and not require that buffers be reused as they are in PIP. SIOs use semaphores (DSP/BIOS II SEM objects) for signaling the TSK that a transfer has completed. This is in contrast to the lower level reader/writer notify functions of PIPs. SIO always uses a semaphore.

PIP and SIO each have their own method of managing data buffers and signaling to the application that a transfer has completed. In the next sections 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 all 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 which 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.

A device driver must also consider system implementation issues such as code size and overhead. If too many features are required then any driver will be too big for a variety of systems. Therefore the interface should be flexible so that peripherals with simple features and requirements will have small, efficient drivers. By the same token the API should allow expansion so that a sophisticated device can be substituted in without rewriting significant pieces of an application.
Another system level issue is resource allocation, that is arbitration of access to on chip peripherals. A mentioned above, a DMA channel is commonly used to transfer data too and from data converters. Often the DMA is used in conjunction with a serial port. Timers may also be used to support periodic sampling. DMA channels, serial ports, and timers, are examples of peripherals that the driver and the application must share. Driver implementations must have a strategy which enables the application and driver to avoid resource contention.

Given that driver users will not always be the driver writers, we need to be concerned with naming conventions to avoid “namespace pollution”. That is we must have a naming convention associated with the drivers so that multiple drivers can be safely used in a single application, and so that drivers can be interchanged without requiring changes to the application.

3 LIO: A Flexible Driver API

Now we describe the API designed to meet the constraints raised in the previous section. This API is called “Low Level I/O”, or LIO. It consists of control functions, I/O buffer management functions, and signaling functions. The functions are shown in Table 1. A driver that implements the LIO functions, called the LIO interface, controls one or more channels on one or more devices.

One aspect to note is that the API is direction agnostic, that is an output-only device implements the same functions as a device which performs input only, or a device that performs input and output. The main difference is between input and output is the sense of the arguments passed to the buffer queue functions. An output channel would be passed a full buffer with putBuf, an input channel would be passed an empty buffer. Since all other operations are identical, much of the control code can be shared between all channels within a single driver.

We'll now present a small set of guidelines and assumptions, outline the LIO functions by category, then provide detailed descriptions for each function. Implementation examples will be shown in the next section.

3.1 General Guidelines, Assumptions, and Calling Conventions

All driver functions should not assume that interrupts are globally enabled. Drivers should not affect the state of global interrupt enable flags. A driver should only affect the state of enable flags for the interrupts triggered by the peripherals it is using. This prevents one driver from starving another driver, or the application itself, for CPU resources.

To avoid namespace collisions due to different drivers using the same function names, and to allow drivers to be changed without recompiling application code, 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, off-chip peripheral, vendor, and the LIO interface. For example, the source code included with this application note implements a DMA-based driver for the TI TMS320VC5402 DSP Starter Kit's AD50 audio codec. Therefore the driver function table name is DSK5402_DMA_AD50_TI_ILIO. The “ILIO” portion stands for LIO Interface, in the style of the eXpress DSP Algorithm Standard. See the TI application note SPRA581.
Each channel supported by a device driver is a half-duplex (either input or output) channel. Each function takes a channel argument. A physical device which can perform both input and output, such as a DSP serial port connected to an audio codec, would be accessed via two half-duplex channels, one for input and one for output. The choice of how many physical devices and channels are supported by a single driver is implementation dependent. Typically a single driver would control a single physical device which may have multiple channels. The mapping of channel numbers to physical device channels is implementation dependent. By convention channel numbers should begin with zero. For I/O devices, even number by convention are input and odd are output. These conventions are not enforced by the LIO interface.

The application should assume that the callback will be called from within an interrupt handler. The driver should assume that the open function will be called prior to global interrupts being enabled. Typically the open function will be called from within the application main function during the DSP/BIOS II startup sequence.

### 3.2 Functions by Type

There are three types of functions in the LIO interface, control, buffer and queue management, and signaling. Let’s discuss each type in detail.

#### 3.2.1 Control Functions

The control functions are used for the startup, shutdown, and control of the device. The init function reserves resources for the driver (both physical peripherals and memory), and programs the device for the expected operation. Init takes a structure pointer as an optional argument. The structure is a device specific argument structure. Fields might be defined for codec sample rate, for example. Other parameters may include channel number for a TDM stream, gain, or color space selection.

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Control</td>
<td>Allocate resources and initialize device.</td>
</tr>
<tr>
<td>Close</td>
<td>Control</td>
<td>Release resources and reset device.</td>
</tr>
<tr>
<td>Cntl</td>
<td>Control</td>
<td>Device specific operations</td>
</tr>
<tr>
<td>Start</td>
<td>Control</td>
<td>Enable the transfer of buffers</td>
</tr>
<tr>
<td>Stop</td>
<td>Control</td>
<td>Disable the transfer of buffers</td>
</tr>
<tr>
<td>GetBuf</td>
<td>Buffer Queue</td>
<td>Retrieve a buffer from the devices output queue</td>
</tr>
<tr>
<td>PutBuf</td>
<td>Buffer Queue</td>
<td>Put a buffer into the device's input queue</td>
</tr>
<tr>
<td>IsEmpty</td>
<td>Buffer Queue</td>
<td>Return true if the device's output queue is empty</td>
</tr>
</tbody>
</table>

Table 1. LIO API Functions
IsFull | Buffer Queue | Return true if the device’s input queue is full
SetCallback | Signaling | Set the function to call when a transfer has completed.

3.2.2 Queue Management

Each device is assumed to hold at least one buffer, that is the buffer being transferred. Many devices, such as a serial port with autobuffering or DMA, also have buffer queues allowing “ping pong” double buffering. Figure 1 shows the three locations in which a buffer may reside within a driver implementing this API. Everything inside the dashed line is considered “inside” the driver. The buffer currently being transferred is typically held in a peripheral register, such as a DMA source or destination register, shown inside the peripheral in the figure. A device with a hardware queue, such as DMA reload registers, will also have one or more locations in which to store pointers for subsequent transfers. This queue is called the “to device” queue. A third location that can contain a buffer pointer is the “from device” queue, or output. This is a variable in the driver itself. The output queue typically contains only one entry, the buffer which was just transferred. A buffer passes from input queue to the peripheral register when the device is ready to transfer a buffer. They buffer then moves to the output queue on completion of a transfer, typically in response to a CPU interrupt.

The queue management functions allow access to these features. putBuf passes a buffer from the application to the driver’s input queue. getBuf retrieves a buffer from the output queue. isEmpty and isFull return the state of the input and output queues. If an input queue is full, then the call to putBuf will return an error because there is no room for the new buffer. If isFull returns false, then a subsequent call to putBuf will succeed. A call to getBuf will return an error code if the output queue is empty because no buffer is available. If isEmpty returns false, a subsequent call to getBuf will succeed. If isFull returns true, then the next call to putBuf may not fail if a transfer completes between isFull returning true and the call to putBuf. A similar race occurs between a true return from isEmpty and a call to getBuf.
Figure 1 shows the buffer queues for a typical device supported by the LIO API. The device has a “to device” queue of buffers to be filled or drained by the peripheral, a single buffer being transferred, and a “from device” queue returning transferred buffers to the application's buffer management.

### 3.2.3 Signaling

As shown in Figure 1, when a transfer is complete a CPU interrupt is typically triggered. This interrupt moves the transferred buffer to the output queue, then calls the callback function passed to the driver by the application. This callback function should perform signaling to the application that the transfer has completed.

### 3.3 Detailed Descriptions

This section describes each LIO function and its arguments in detail. The functions are discussed in alphabetical order. The return and argument types use the DSP/BIOS II standard typedefs provided in the DSP/BIOS II std.h C include file.

#### 3.3.1 Void close(Uns chan)

Close deallocates any resources reserved for a channel when it was opened. Unlike the stop function described below, any transfers currently in progress will be immediately halted. All buffers in queues will be ignored, it is up to the application to free any buffers currently in the queues. It is an error to call close for a channel which has not been opened, the behavior is undefined in this case.
3.3.2 \textit{Int ctrl(Uns chan, Uns command, Arg arg)}

Perform a device specific control operation on a channel. The operation to be performed is selected by the command argument, and the mapping of command argument values to operations is completely device specific. Possible operations include selecting data formats, or setting device parameters such as codec sample rate.

3.3.3 \textit{Ptr getBuf(Uns chan, Int *size, Int *dataSize)}

Remove a processed buffer from the from-device queue. This is the complement operation to putBuf. If the value of the chan argument corresponds to an input channel then the buffer has been sent. If the value of the chan argument corresponds to an input channel then the buffer contains received data. The return value is a pointer to the processed buffer. If no buffer was available then NULL is returned. The size and dataSize arguments are pointers to integers. The size argument will be assigned the size of the buffer returned. The dataSize argument will be assigned the number of bytes [footnote: bytes are 16 bits on 54x and 55x processors] of valid data in the buffer. For a sent buffer, this will be the number of bytes actually sent. For a received buffer this will be the number of bytes received.

3.3.4 \textit{Bool isEmpty(Uns chan)}

Return the state of the from-device queue. If isEmpty returns TRUE, then there are not buffers ready to be returned via the getBuf call. If isEmpty returns FALSE, then there is at least one buffer ready to be retrieved via getBuf. Think “Is the device driver empty?”.

3.3.5 \textit{Bool isFull(Uns chan)}

Return the state of the to-device queue. If isFull returns TRUE, then a call to setBuf will fail because there is no room in the device driver queue for another buffer. If isFull returns FALSE, then a call to setBuf will succeed. Think “Is the device driver full?”.

3.3.6 \textit{Int open(Uns chan, Ptr args)}

Prepare a channel for use. Open must be called for a particular channel number before any other call can be made for that channel number. Typically this call will reserve resources and initialize peripheral registers to make the device ready to transfer data. The args argument is a pointer to a device specific parameter data structure. It may be set to NULL to choose default parameters or if the device has no parameters. If the channel is successfully opened, then zero is returned. If an error occurs a non-zero value is returned. –1 indicates a general error opening the channel. Other values are driver specific.

Open may be called by the application with interrupts globally disabled, so channel initialization should not depend on the global interrupt state. For example open may be called from the main function of DSP/BIOS II application. See the \textit{DSP/BIOS II User’s Guide} for details of the processor state during the call to main from the DSP/BIOS II startup sequence.
3.3.7 **Bool putBuf(Uns chan, Ptr buf, Uns size)**

Place a buffer on the to-device queue. This is the complement operation to getBuf. TRUE is returned if there was room on the to-device queue, FALSE otherwise. The Buf argument is a pointer to the buffer, size is the size of the buffer in bytes. If chan is an output channel then the buffer contains data to be sent. If chan is an input channel then the buffer is empty and the contents will be overwritten by the driver.

3.3.8 **Void setCallBack(Uns chan, LIO_TcallBack cb, Arg arg)**

Set the application function to be called when a buffer transfer has been completed. The cb argument is a function pointer, the function type signature is defined as:

```c
typedef Void (*LIO_TcallBack)(Uns chan, Arg arg);
```

The arg argument to setCallBack is passed as the arg argument to the callback. The application should assume that the callback will be called from within an interrupt handler.

3.3.9 **Void start(Uns chan)**

Start the flow of data. Start must be called after open or no data will be transferred. Start should not affect the state of any global interrupt flags, therefore data transfer may not actually begin until the application enables global interrupts. Start may be called multiple times without additional effect.

3.3.10 **Void stop(Uns chan)**

Stop the flow of data. The complement call to start. Data does not stop immediately, the driver may completes a buffer transfer in progress before halting. For example, a video frame buffer driver will complete transferring a full image to a display. If a transfer is in progress when a call to stop is made then the callback will still be called when the transfer completes. Stop may be called multiple times with no additional effect.

4 **Writings LIO drivers: Implementation Examples**

In this section we explore the implementation of two different device drivers connected to a similar peripheral. Audio processing, such as voice compression and call progress tone detection, are common applications of digital signal processors. In this section we look at the implementation of the LIO interface for two different Texas Instruments DSP Starter Kits, the TMS320C6211 DSK and TMS320VC5402 DSK.

The first example is a sample-by-sample interrupt driver using the McBSP serial port on the TMS320C6211 to send and receive data from the 6211 DSK audio codec. DMA is not used in this driver for simplicity. The second example uses DMA to transfer through the McBSP on the TMS320VC5402 to and from the audio codec on the 5402 DSK. Complete source code is provided with this application note.

Both implementations have a similar architecture. The driver is set up as a state machine. A C data structure is used to track the state of the driver as it responds to application calls and device interrupts. The state variables are the state of the device driver buffer queues shown in Figure 1 on page 7. Figure 2 below shows the simplest set of state transitions in this model.
The letters in the circles are the state of the device driver buffer queues. The first letter is the to-device queue, the second is the buffer held by the peripheral, and the third is the from device queue. E means empty, F means full. EEE is the start state, entered after a call to open.

![Diagram of state transitions]

**Figure 2. Simple LIO Implementation State Transitions**

Each circle is a state, and the letters are the status of the device driver buffer queues. The first letter is the status of the to-device queue, the middle is the buffer pointer held by the peripheral during a transfer, and the last letter is the from device queue. Each queue may be empty (E), full (F), or not empty nor full (N). In Figure 2 only E and F queue status conditions are seen. In the initial state on the left the driver is empty, state “EEE”. The application calls putBuf to place a buffer on the to-device queue. The driver places that buffer immediately in the peripheral and transitions to state “EFE”. When the transfer is complete an interrupt is signaled by the peripheral to the driver. The driver’s interrupt handler then moves the buffer from the peripheral register to the from-device queue, transitions to state “EEF”, then calls the application’s callback function. The callback calls getBuf to retrieve the buffer from the driver’s from-device queue, and the driver returns to the start state.

If the driver supports hardware queuing, then the to-device queue can hold a buffer while another is being transferred by the peripheral. In Figure 3 below, the state transitions are shown with one level of queuing. As in the previous example, the start state is “EEE”. The application adds a buffer to the to-device queue, and the device driver immediately moves the buffer into the peripheral buffer pointer location and moves to state “EFE”. While in this state the application may now add another buffer to the to-device queue. The driver saves this buffer pointer in a queue, and the state is now “FFE”, where the to-device queue is full, the peripheral is transferring a buffer, and the from device queue is empty.

The software implementation of this state machine uses a C data structure to contain the state vector described above. LIO API calls and interrupts cause transitions in the state graph, that is a change to the state structure. The next two sections describe implementations of this state machine, one without DMA and one with.
4.1 6211 DSK: Sample-by-Sample with McBSP

The simplest implementation of an LIO driver performs sample by sample transfers, that is an interrupt is signaled by the device for every data point to be transferred. Using the on-chip EDMA would be more efficient, the sample-by-sample method is used here for simplicity. The interrupt handler must determine when a full buffer has been transferred, and perform the appropriate buffer queue movement. In this example an LIO driver has been implemented which supports input and output between the TMS320C6211 and TI ad535 audio codec on the TI TMS320C6211 DSK. The ad535 is physically connected to a multi-channel buffered serial port, or McBSP, on the 6211. The McBSP is configured by the driver open functions to generate an interrupt when a new word has been received and when a word has been transmitted.

Figure 4 shows the driver state structure which is operated on by the LIO API functions and interrupt handler. The first field, running, is a boolean that is operated on by the start and stop LIO API functions. This allows critical sections in the driver code to know whether or not to reenable interrupts. The next fields, currentBuffer and currentSize, hold the start address and size of the buffer being transferred, the current buffer. This is the pointer to be moved to the from-device queue when the transfer is complete. The next two fields, currentPointer and currentSize, are used by the interrupt handler to transfer the next sample and detect the end of the current buffer. The fullBuffer and fullSize fields implement a one-deep from-device queue. NextBuffer and nextSize implement a one-deep to-device queue. The callback function address and argument are also stored in the state structure by the setCallback function.
typedef struct drv_state {
    Bool running;      /* Should interrupts be enabled? */
    Ptr currentBuffer; /* Saved pointer to buffer being filled or drained */
    Uns currentSize;   /* Size in MAUs of that buffer. */
    Uns currentPtr;    /* Pointer to next insertion point, used by ISR. NULL if no buffer 
                        is currently being transferred. */
    Uns currentCount;  /* Count of elems in currentPtr(NOT MAUs), used by ISR */
    Ptr fullBuffer;    /* Previously filled or drained buffer, to be returned by getBuf */
    Uns fullSize;      /* Size of fullBuffer */
    Ptr nextBuffer;    /* Next buffer to fill or drain after currentBuffer */
    Uns nextSize;      /* Size of nextBuffer */
    LIO_TcallBack callback; /* Callback to notify of new full buffer */
    Arg callbackArg;
} LIO_Obj;

Figure 4. Sample-by-Sample State Structure

The driver contains an array of these LIO_Obj, one for each channel. As implemented on 6211 
DSK the ad535 supports one input and one output channel of 8khz 16 bit audio data, so the 
driver contains an array of two LIO_Obj. We will call this array chans. By convention channel 0 
is the input channel, and channel 1 is the output channel.

Figure 5 shows the implementation of the putBuf LIO API function, and how it manipulates the 
fields of the LIO_Obj within the chans array. The first step is to disable the McBSP interrupt if it 
is enabled. This provides mutual exclusion for the state structure between the application and 
the interrupt handlers. Note that no mutual exclusion is provided for multiple application threads, 
that is up to the application to provide if needed. The next step is to determine if a transfer is 
already in progress. PutBuf has been passed a new buffer by the application for transferring. If 
no buffer is currently being transferred then the buffer is immediately placed in the “peripheral” 
register, or the currentBuffer field of the state structure, and the interrupt handler is kicked 
off. If a transfer is already in progress, then the to-device queue is checked. If it is empty the 
buffer is placed there and putBuf returns a success code. If the to-device queue is full then a 
failure code is returned. Before returning on any path, the McBSP interrupt is reenabled if the 
running field is set. That check is encapsulated in the local subroutine enableIntr.

The interrupt handler is called when a word has been received from the codec or a transmitted 
word has been sent to the codec. The interrupt handler then reads or writes the next word, and 
checks to see if the all words of the buffer have been transferred. If so it moves the contents of 
the currentBuffer and currentSize state fields to the fullBuffer and fullSize fields, then checks to 
see if there is a buffer in the to-device queue (nextBuffer field of the state object). If so, it 
moves that buffer into the currentBuffer, currentPtr, currentSize, and currentCount fields. Then 
the callback is called.
static Int putBuf(Uns chan, Ptr buf, Uns size)
{
    /* Make sure nothing changes while we're changing state. Note that
     * this assumes that only an ISR and one user thread are accessing
     * this channel. If more than one user thread needs to access the
     * channel then the application must implement mutual exclusion. */
    disableIntr(chan);

    /* Are we filling a buffer already? */
    if (chans[chan].currentBuffer != NULL) {
        /* yes we are. Do we have room for a nextbuffer? */
        if (chans[chan].nextBuffer == NULL) {
            /* yes */
            chans[chan].nextBuffer = buf;
            chans[chan].nextSize = size;
            retVal = 1;
        } else {
            /* no room, return failure */
            retVal = 0;
        }
    } else { /* No current buffer, set it up and go! */
        retVal = 1;
        chans[chan].currentBuffer = buf;
        chans[chan].currentSize = size;
        chans[chan].currentPtr = (Uns)buf;
        /* Units of count is elements, not MAUs */
        chans[chan].currentCount = size / sizeof(short);

        /* If receiving, clear old data from MCBSP if present */
        if (chan == RCV_CHAN && MCBSP_Rrdy(hMcbsp)) {
            MCBSP_Read(hMcbsp);
        }
    }

    /* Kick it off again */
    enableIntr(chan);
    return retVal;
}

Figure 5. Implementation of putBuf LIO API Function for Sample-by-Sample Driver

Note that the putBuf function shown in Figure 5 is declared static. The application calls the
function through a function table, which is global. The table is shown in Figure 6. The interface
table is a C data structure with an entry for each LIO function. The concept is identical to the
interface tables used by the eXpress DSP Algorithm Standard defined in TI application note
SPRA581. The naming convention ensures that two drivers will never have the same global
symbol for their function table. This means that drivers from different vendors will be able to be
linked into the same system without name conflicts. This interface table should be the only
global symbol visible. From the name shown in Figure 6, we see that this driver is for the
DSK6211 board, uses an on-chip McBSP for connecting to an off-chip AD535, that Texas
Instruments is the vendor, and that this table implements the LIO interface.
4.2 5402 DSK: DMA with Reload

In this example we use the DMA on the TI TMS320VC5402 DSK to move audio codec data from a McBSP into our buffers. The DMA registers will hold the currentPtr and currentCount variables from the previous example. We will also use the DMA global reload registers to hold the to-device queue, so we no longer need the nextBuffer and nextSize fields of the state structure. The state structure for this driver is shown in Figure 7.

```c
typedef struct drv_state {
  Bool enabled;  /* Have we been started or are we stopped? */
  Ptr currentBuffer;  /* Saved pointer to buffer being filled or drained */
  Uns currentSize;  /* Size in MAUs of that buffer. */
  Ptr fullBuffer;  /* Pointer to just completed buffer */
  Uns fullSize;  /* Size in MAUs of that buffer. */
  LIO_TcallBack callback;  /* Callback to notify of new full buffer */
  Arg callbackArg;
} LIO_Obj;
```

This driver only receives a single interrupt per buffer, instead of an interrupt per sample. When this interrupt occurs the driver already knows that a buffer transfer has completed, and since reload is being used the DMA does not need to be reprogrammed. This interrupt handler first moves currentBuffer to fullBuffer. Next, if a buffer was in the to-device queue, that is DMA reload is being used, then the new buffer pointer and size are recorded in the currentBuffer field. The callback is then called as before.

As you can see once the basic state machine structure has been defined, new drivers for similar hardware are easy to write. As well DMA becomes an easy feature to support where appropriate. In the next section we will show how to use an LIO driver from within an application.
5 Using LIO Drivers: Application Examples

The first section of this application note presented a set of application requirements for device drivers to be used with DSP/BIOS II. We then presented a design, and two implementations, of a particular device driver model called LIO. In this section we demonstrate the use of LIO with DSP/BIOS II I/O and signaling primitives. In the first example we show using the LIO interface directly from an application, that is using it raw. The application does a loop-back, copying the data coming in back to the output, for simplicity. The next section shows how to construct and adapter from the DSP/BIOS II PIP object to the LIO interface. This adapter can be used with any LIO driver to support the use of PIPs in your own application. Again we construct a loop-back example using PIP. Finally we round out the examples with an adapter using the DSP/BIOS SIO module with an LIO. Complete source code for these examples, including the PIP and SIO adapter code, is provided with the application note or on the Texas Instruments web site. Code excerpts and summaries are presented here for discussion.

5.1 Raw

This application drives the LIO driver state machine directly from the LIO API. The data buffers are static arrays, and signaling is done with a DSP/BIOS II semaphore, or SEM, object. The semaphore could be replaced with a done flag and a while loop, the function here is the same.

```c
/* Set DRIVER_FXN_TABLE in the project options or linker command file. */
extern LIO_Fxns DRIVER_FXN_TABLE;
LIO_Fxns *driver = &DRIVER_FXN_TABLE;

#define RCV_CHAN 0
#define XMT_CHAN 1

Void main()
{
if (!driver->open(RCV_CHAN, NULL)
    || !driver->open(XMT_CHAN, NULL)) {
    LOG_printf(&trace, "init failed");
} else {
    driver->setCallback(RCV_CHAN, callback, 0);
    driver->setCallback(XMT_CHAN, callback, 0);

    driver->putBuf(RCV_CHAN, buf0, BUFSIZ);
    doubleBuffered = driver->putBuf(RCV_CHAN, buf1, BUFSIZ);
    if (!doubleBuffered) {
        LOG_printf(&trace, "Driver has no hardware queue.");
    }
    driver->start(RCV_CHAN);
    driver->start(XMT_CHAN);
}

Figure 8. Main Routine from raw.c
```
/** First include DSP/BIOS II generated cmd file */
-l rawcfg.cmd

/** Bind low level driver */
 DRIVER_FXN_FXNS=_DSK6211_MCBSP_AD535_TI_iLio;

Figure 9. Linker Command File for Binding the Driver Function Table

Figure 8 shows the main routine from the raw example. In main, the receive and transmit channels are opened by directly calling the driver open function. NULL is passed as the options argument, the default settings for the drivers in these examples are 8khz sample rate and 16 bit data. Observe that the variable driver is declared as a pointer to an LIO function table. The actual table symbol, DRIVER_FXN_TABLE, is assigned in the linker command file to the driver to be used. This example is completely platform independent. For the DSK6211, the DRIVER_FXN_TABLE symbol will be assigned to the DSK6211_MCBSP_AD535_TI_iLio LIO interface table defined in Figure 6. Figure 9 shows the linker command file which does the binding. The first line includes the linker command file produced by the DSP/BIOS II configuration tool, which places the code and data sections of the application. The last line assigns the function table to the symbol DRIVER_FXN_TABLE, creating the driver binding for this application.

Before starting the device putBuf is called twice to double buffer the input. This example checks the return value from the second call to putBuf to see if the device indeed supports double buffering. If not a flag is set that will be used below. Finally we start the channels. Note that this is called from the main routine of a DSP/BIOS II application, so global interrupts have not yet been enabled. The device will actually start functioning after main returns.

5.2 PIP With SWI

The DSP/BIOS II 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 II 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 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. The notify functions associated with a PIP object are selected by the user in the DSP/BIOS II configuration tool.

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 results in the readerNotify 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. This triggers the writerNotify function, and the cycle begins again. To use an LIO driver with a PIP object, we must create a writerNotify function that will fill the PIP from an LIO input channel, and a readerNotify function that will drain a PIP into an LIO output channel. We will call these functions a “PIP adapter”, a set of functions that connects the PIP API to the LIO API.
The example we present here is again a loopback program. It is identical in function the DSP/BIOS II Audio example included with Code Composer Studio, however the audio example does not use an LIO driver. This example, which we'll call pip_echo, will demonstrate how to write a PIP adapter and how to configure a DSP/BIOS II PIP object to use the adapter and an LIO driver. First we will describe the adapter implementation, then the use. The adapter described here can be used with any LIO, and in any application. You may use this adapter in your own applications.

5.2.1 The PIP Adapter

The pipe adapter design used in this example implements a state machine that matches the states of a PIP to the states of an LIO. Figure 10 shows the possible states of a two-frame PIP. The circles are the states where the state vector is the number of frames available to the writer, number of frames available to the reader, and the state of PIP_alloc/PIP_put, PIP_get/PIP_free, and PIP.peek operations. The transitions are taken on calls to the PIP API. The names have been abbreviated here for clarity, PIP.alloc is simply alloc, etc. See the DSP/BIOS II Users Guide for complete documentation on the PIP module.

Figure 10. PIP Access State Diagram

The black states show the most common path through the possible states of a two-frame PIP. The gray states are also valid, but typically not used in a double-buffered real-time system.
In the start state on the left, there are two frames available to the writer side of the pipe. PIP_alloc is called to retrieve a frame to write data into. At this point a call to a device driver’s putBuf call can be made to request the driver fill the buffer. Because of the implementation of PIP within DSP/BIOS II, PIP_alloc can only be called once before PIP_put is called. Therefore at this state the application must wait for the driver to fill the frame before calling PIP_put and passing the frame to the reader side of the PIP. Since we have not called PIP_put yet, we cannot add another buffer to the to device queue by calling PIP_alloc. However, the PIP.peek function can be called to get the address that the next call to PIP_alloc will give. The state diagram shows this transition to state “1,0,PA”, where the P indicates that a peek has been performed. When the transfer is finished, the frame just filled is passed to the reader side of the PIP with PIP.put, and the alloc of the next frame is performed. Since we performed a previous peek, this PIP.alloc does not need to actually make an LIO call.

Now we have called PIP.put, so there is a buffer available to the reader side of the PIP, and the readerNotify function of the PIP is called. The reader will call PIP.get, process the buffer, then call PIP.free to give the now empty buffer back to the writer side. If this is done before the second buffer is filled then the call to PIP.free takes the PIP back to the “1,0,A” state. This path is shown in black in Figure 10. If the processing of a buffer takes less than the time for another buffer to be received, then this black path will be taken and real-time processing of the input data is achieved.

How do we interface this state machine to LIO? Every call to PIP_alloc in the PIP state diagram will be followed by a call to LIO_putBuf to an input channel for the underlying LIO. Every PIP_put transition is taken in response to a callback from that channel and is performed after a call to LIO_getBuf to remove the just-filled buffer from the driver. So the writerNotify function performs PIP_alloc or PIP.peek and LIO_putBuf, and the callback performs a LIO_getBuf and PIP_put. When an LIO is connected to the reader side of a PIP, the readerNotify does at PIP_get followed by an LIO_putBuf, so the relationship of get and put is reversed. The accompanying source code implements this functionality.

5.3 SIO With TSK

The DSP/BIOS II Streaming I/O, or 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 II printed documentation 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, however, because they require a higher level of DSP/BIOS II knowledge, specifically the writer must know the QUE, SEM, and SIO modules in order to implement an SIO DEV driver. As well a DEV driver can only be used with an SIO, it cannot be used with a PIP or on it’s own.

In this example we implement a DEV which 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 implementation is given in the source code examples.
6 Multichannel I/O

So far we have focused on single channel I/O devices, that is from the application point of view our LIOs have been used within a single thread. However many peripherals are multi-channel peripherals. Example include stereo audio codecs and time division multiplexed data streams such as T1/E1 streams. With a typical multi-channel peripheral, the data is interleaved for all channels within a block transfer. In other words the order in which the data is sent or received starts with the first word of the first channel, then the first word of the second channel, until the first word of the nth channel. Then the second word of the first channel, and so on. The DMA units on most Texas Instruments DSPs will de-interleave such data streams so that all the data from a particular channel is contiguous, and all channels buffers are contiguous. Figure 11 shows this data flow from a TDM stream, through a McBSP and DMA configured for de-interleave into a memory block.

The LIO interface, however, takes buffers from anywhere in memory, and they may not be synchronized on block boundaries. In order to support LIO for these types of systems, the LIO driver must move data from the contiguous channel blocks into the application data buffers, as show in Figure 12. DMA channels can be used to perform these moves, or they can be done with the CPU. This type of two-level architecture is common in multi-channel “universal port” telecom systems, where each TDM channel may be transferring data units of different size corresponding to different voice compression algorithms.

Figure 11. Multichannel I/O using DMA

The DMA is typically configured to sort the interleaved data into blocks within a frame.
DMA Buffers

| Ch1   | 10ms buffer |
| Ch2   | 12ms buffer |
| Ch3   | 20ms buffer |
| Ch4   | 12ms buffer |

LIO

Figure 12. A Multichannel LIO Scatter Driver

This LIO uses memory-to-memory DMA to move de-interleaved data into application buffers.

7 Overhead

Most of the functionality of LIO must be implemented in any system that does not use fixed location memory buffers for I/O. Given that, the question of overhead revolves around the function table and callback mechanisms. Using the function table calling method virtually eliminates the problem of name collisions. However it takes two memory loads in order to make a function call, one load for the table address, then another to load the function address. A third instruction is then required for the function call itself. If direct linking to the functions was used each call would as little as one instruction. Similarly the callback requires and indirect call from within the interrupt handler and a return, while a custom driver would simply perform the operations in the callback directly.

We can see that even though each function call takes three times longer, these calls are very infrequent. Consider a typical 8KHz 20ms buffer from a telecom application. Each frame requires a call to putBuf, getBuf, and the callback. In a custom system this would require as little as $1 + 1 + 0 = 2$ cycles per frame. In an LIO system it would require approximately $3 + 3 + 3 = 9$ cycles per frame. At 50 frames per second, this is 450 instructions versus 100 instructions. 350 instructions per second is only 0.00035 extra MIPS on a 100MHz TMS320 DSP.

8 Conclusion

This application note has defined a low level frame-based streaming I/O driver interface which can support 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. Three different applications which use the same drivers have been shown, and adapters provided for use with DSP/BIOS II 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 II.
IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 2000, Texas Instruments Incorporated