Application Report
SPRA689 - October 2000

Using DSP/BIOS I/O in Multichannel Systems

Don S. Gillespie
Digital Signal Processing Solutions

ABSTRACT

Digital Signal Processors (DSPs) are typically used to input large amounts of data, perform mathematical transformations on that data, and then output the resulting data, all at very high rates. While some applications contain only a single channel of input data and a single channel of output data, many applications are more complicated in that they consist of multiple channels of input and output data, some at differing rates of speed. Some examples of multichannel applications are telecommunications (voiceband and baseband), sensors for medical equipment (EKG), optical sensors, motion sensors, and audio devices. The detailed processing performed on the data is specific to the individual application. But most multichannel applications share certain characteristics: they need to input data in a deterministic manner, perform some kind of processing on it, and output some results in a deterministic manner, while meeting the specific rate and priority requirements for that data.

DSP/BIOS II affords the application designer multiple ways to achieve the requirements of a multichannel system. PIPs and SIOs provide two different approaches to achieving device-independent I/O. Each approach carries its own set of advantages, disadvantages, features, and cost. Additionally, there are multiple ways to use each approach. This application note will compare and contrast these approaches, from the aspect of single-threaded as well as multi-threaded designs.

Contents

1 Introduction 2
1.1 General Notes on Trade-offs 3
2 SIO 3
2.1 Using a Single Task 3
2.1.1 Single Task, Standard Model 3
2.1.2 Single Task Using SIO_select 6
2.2 Using Multiple Tasks 7
2.2.1 Using a Single Task Per Channel 7
3 PIP 11
3.1 Using a Single SWI Per Channel 13
3.1.1 notifyReader and notifyWriter Routines 16
4 Summary 18
5 References 18

List of Figures

Figure 1. SIO With Single Task 3
Figure 2. Initialization in Example sio1 4
Figure 3. Processing Loop in Example sio1 5
Figure 4. Execution Graph for Example sio1 6
Figure 5. Array of Stream Handles in sio2 7
Figure 6. SIO with Single Task per Channel 7
Figure 7. Execution Graph with Different Loads Per Channel 8
Figure 8. Priming of Output SIO in Example sio4 10
Figure 9. Processing Input Stream in Example sio4 11
Figure 10. PIP API Calls and Callbacks 12
Figure 11. Data and Control Flow: Single SWI PIP Example 15
Figure 12. notifyWriter Pseudo-code for Receive 16
Figure 13. ISR Pseudo-code for Receive 16
Figure 14. notifyReader Pseudo-code for Transmit 17
Figure 15. ISR Pseudo-code for Transmit 17

1 Introduction

The bottom line for any designer of a multichannel system is that the streams of data moving through the system must be handled quickly, deterministically, and correctly. Additionally, a scheme for such data movement should be easy to use, and should not have to be re-written or ported for each new system designed. DSP system designers have dealt with the issues involved with multiple data streams for a long time, usually by writing their own drivers, buffer-managing schemes, and schedulers. One problem with this approach is that for each new system, the basic design must be modified, ported, or completely re-written to meet the new requirements. These new requirements may involve more channels of data, or different transfer parameters on different streams. With these considerations, writing your own stream-handling code is not very efficient in the long term.

DSP/BIOS II offers two basic constructs for handling data streams in a real time system: Streaming I/O (SIO), and Data Pipes (PIP). They differ in their approach to solving the problem, but they both provide known, solid, deterministic methods of handling data streams regardless of the number of channels. Additionally, porting is not an issue, since the DSP/BIOS II kernel is available for all C6000 and C54x devices.

This note presents SIO and PIP objects from the point of view of designing multichannel systems. Code examples are included and discussed, to highlight the different features of each method. The code examples are provided in a zip file which accompanies the note. The goal is to provide the designer with a clear understanding of how to use SIO and PIP, and also to provide him with multiple design choices. The code examples are simple, two-channel applications which copy audio data from input to output. They utilize the same hardware setup as the DSP/BIOS II audio example (detailed in the application note *An Audio Example Using DSP/BIOS II*, SPRA598). This simple application was chosen so that the architectural issues of the different methods can be more easily highlighted. In addition, the code examples may be easily modified and used as a basis for system designs.

The code examples run on both the C6201EVM and the C6211DSK. Note, however, that on the DSK, only one audio channel will be audible because the DSK supports only one channel.

In order to get the most out of the code samples, you must be able to open the example project files, configuration files, and source files in CCS while you are reading this note.

When running the DSK examples in CCS 1.20, some error assertions may appear in the execution graph, accompanied by "invalid thread state transition" errors in the system log. This is not actually an error; as you will notice the application is not affected. These assertions will not occur in future versions of CCS.

1.1 General Notes on Trade-offs

In general, SIO objects incur more overhead in processing time and memory usage than do PIP objects. In addition, TSK objects incur more overhead than SWI objects.The cost for this increased efficiency is some increase in complexity and less flexibility. However, for many applications, TSK and SIO are adequate. Efficiency is also affected by other system-specific factors: number of channels, whether objects may be static or dynamic, amount of memory available, number of different interrupts, rate differences, number of algorithms, efficiency of the device drivers. All of these factors together drive the choice for a particular architecture for a particular application.

The timing overhead incurred in each API call is detailed in application note SPRA662, *Benchmarking DSP/BIOS II on the C6000*.

2 SIO

The examples using SIO start with the most simple, basic approach, and gradually build in complexity.

2.1 Using a Single Task

The most simple scenario to consider for a multichannel system using SIO is that of a single task handling both channels of data, as shown in Figure 1.

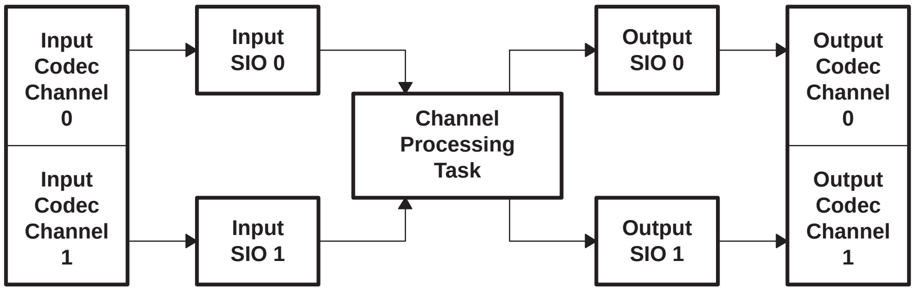

Figure 1. SIO With Single Task

There are two channels of data streaming in. They are each handled by a different SIO. The data passes through the two input SIOs to the task, which will perform the necessary processing. The task then passes the data to two different output SIOs, which in turn pass the data through to the output hardware.

The code examples demonstrate two ways to implement this architecture.

2.1.1 Single Task, Standard Model

The first model in the directory /ti/myprojects/spra689/examples/*board_type*/examples/sio/sio1 (where *board_type* is dsk or evm) shows the most basic implementation. This architecture has some pitfalls, which will be explained later.

The file that contains the main application code is echo.c. The main task in this application is *echoTSK*. As you can see in the DSP/BIOS II configuration, echoTSK consists of the routine *echo*. In echo, you will see some initialization code followed by an infinite loop (while (TRUE)). This is the main processing loop of echoTSK.

In the DSP/BIOS II configuration, there are four SIO objects: *inStream0*, *inStream1*, *outStream0*, and *outStream1*. They are all defined as the *standard SIO model*. In the standard model, the APIs SIO_get and SIO_put are used to interact with the data stream. Each stream has two buffers defined; these are our ping-pong buffers. They are the buffers into which the audio data will be loaded. The switching between buffers is accomplished by the SIO module; the application does not have to worry about the details.

Although the examples in this note use SIOs that are statically defined in the DSP/BIOS II configuration, SIOs may also be created and deleted dynamically via DSP/BIOS II APIs. This gives the designer the freedom to create and delete SIOs on the fly if the required number is not known at compile time.

A key element to understanding the SIO standard model is that every interaction with the stream is a buffer exchange. No buffer is ever given to the stream or taken from the stream without exchanging it with another buffer. If no buffer is available for exchange, the task will pend within the API until a buffer is available, causing a context switch. This exchange enables the application to process one buffer while the SIO module is filling or emptying another.

For each stream defined in the configuration, the two buffers are allocated by the SIO module; the application does not have to MEM_alloc them, or MEM_free them. There is only one buffer the application must create; that buffer is used in the first SIO_get, to begin the streaming process. Before the first SIO operation on a stream, the stream has two buffers already. Our application will call SIO_get to request a full input buffer. However, all SIO operations in the standard model are exchanges, so in order to get a buffer full of data from the stream, we must pass it an empty buffer. That is why echo creates a buffer as part of its initialization; so that it may be passed to the stream so that a full buffer can be retrieved. All applications that use the standard model of SIO must create this initial buffer. MEM_alloc is one way to create the buffer, but the method used in the example is to use a static buffer created by an SIO. The SIO inStream0 has *the Allocate Static Buffers* box checked. At the start of the echo routine, SIO_staticbuf is called to get the address of the static buffer that the application will use to prime the SIO on the first SIO_get. See Figure 2.

```
/* initialize the channel database */
      audioChannel[CHANNEL0].inStream = &inStream0;
      audioChannel[CHANNEL0].outStream = &outStream0;
      audioChannel[CHANNEL0].load = 0;
      audioChannel[CHANNEL1].inStream = &inStream1;
      audioChannel[CHANNEL1].outStream = &outStream1;
      audioChannel[CHANNEL1].load = 0;
      /* create the buffer used to prime the input stream */

 if (SIO_staticbuf(audioChannel[CHANNEL0].inStream, &buf) == 0) {
     SYS_abort(”Cannot get static buffer”);
 }
```

Figure 2. Initialization in Example sio1

Notice that in the calls to *processChannel*, where the actual SIO operations take place, the buffer pointer is not passed; rather, a pointer to the buffer pointer is passed. See Figure 3. This pointer is passed on to the SIO_get and SIO_put APIs. When the SIO_get and SIO_put calls each return, the pointer passed in will point to a different buffer than it did when the call was entered. This is how the buffer exchange is accomplished. During the execution of the application, the value of the local variable *buf* in the echo routine continuously changes as a result of every call into the SIO API. To see this, load the code to the target, open a DSP/BIOS II Message Log, and set it to *trace1.* Run the example, and examine the message log closely. The value of the buffer pointer before and after every SIO API call is logged. Notice that the returned values are different from the values passed in. Also, notice the pattern in buffer addresses. There are nine buffer addresses that continuously cycle. The configuration has four streams defined,three with two buffers and one with two plus a static buffer, making nine total. We can conclude from this that buffers created in the definition of an SIO object can be passed to any SIO, not just the one where it was declared.

```
/* loop forever: first process channel 0; then process channel 1 */
while (TRUE) {
      LOG_printf(&trace1, ”channel 0”);
      if (!processChannel(&audioChannel[0], &buf)) {
            LOG_printf(&trace, ”processing error on channel 0”);
      }
      LOG_printf(&trace1, ”channel 1”);
      if (!processChannel(&audioChannel[1], &buf)) {
            LOG_printf(&trace, ”processing error on channel 1”);
      }
}
```

Figure 3. Processing Loop in Example sio1

Let’s take a closer look at the contents of the *processChannel* routine. It consists of an SIO_get to an input stream, and an SIO_put to an output stream. Between these two operations is where any actual processing of the input data would go. To simulate this processing, there is a load function called with a load value. To use this value to simulate a processing load, display the audioChannel array in the watch window. Edit the load value for either channel, or both. Each channel has its own load value, but since they are processed in the same task, additional load will be seen on the Execution Graph only as increased processing time for echoTSK. See Figure 4. When the load(s) are increased high enough, the audio quality will degrade as the application starts missing its real time deadline and begins dropping frames. This can be seen by viewing the _trace log, or system log.

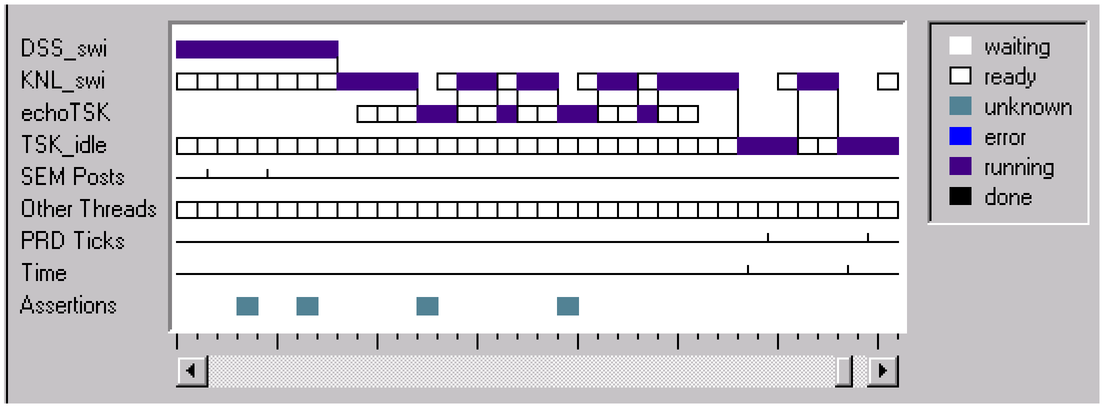

Figure 4. Execution Graph for Example sio1

Each SIO_get and SIO_put actually performs an *issue*, and a *reclaim* at the device driver level. The issue is performed first, and passes the input buffer pointer to the stream. In the case of SIO_get, this is the empty buffer; in the case of SIO_put, the buffer is full. The next operation is a reclaim. This operation returns a buffer pointer from the stream. It is this operation that will pend, in the event that no buffer is available for return.

The problem with this architecture is that it assumes the order in which the channels should be processed, and hardcodes that order. If the program counter is at the top of the loop in echo, but channel 1 is ready for input processing, channel 0 processing must complete first in order to process channel 1, because that is how it is coded. If echoTSK pends in either SIO operation, the wait will be even longer. Also, if channel 0 stops functioning completely, the task will pend forever, unless a specific timeout is specified in the SIO configuration. Still, this architecture may be adequate for systems where all channels are running at the same rate, and undergoing processing by the same algorithm.

2.1.2 Single Task Using SIO_select

The code example for SIO_select is found in the /ti/myprojects/spra689/examples/*board_type*/examples/sio/sio2 directory (where *board_type* is dsk or evm). The functionality of the example is the same: two audio channels are input, copied, and output. But neither channel is actually processed until there is a buffer available for processing. This ensures that the next SIO operation performed on that stream will not pend.

SIO_select takes as input an array of stream handles to check for readiness, the number of streams to check, and a timeout value. With a timeout of 0, SIO_select will exit immediately after checking the streams, even if no streams are ready. With a valid timeout, it will cause the calling task to pend until at least one of the specified streams becomes ready or the timeout value is reached. SIO_select returns a mask indicating which streams are ready. See Figure 5. The application then tests the mask to determine which streams to process. This method differs from the previous one in that it handles the channels as they become ready, not necessarily in a predetermined order.

```
SIO_Handle        inStreams[NUM_CHANNELS];

/* SIO_select requires an array of the streams it checks for input; this */
/* initializes that array */
inStreams[0] = &inStream0;
inStreams[1] = &inStream1;
/* loop forever: check each input stream for input; only perform processing
*/
/* for a channel if the channel has input ready */
while (TRUE) {
      mask = SIO_select(inStreams, 2, SYS_FOREVER);
```

Figure 5. Array of Stream Handles in sio2

This architecture works well for streams of differing rates or types of data, such as may occur in a multiprocessing system; or simply as a more flexible alternative to the previous architecture.

2.2 Using Multiple Tasks

2.2.1 Using a Single Task Per Channel

As seen in Figure 6, the single task per channel approach separates the processing for each channel into individual tasks. This is a more modular approach, which simplifies the processing. Now a task can focus on only one channel and its processing, instead of two or more.

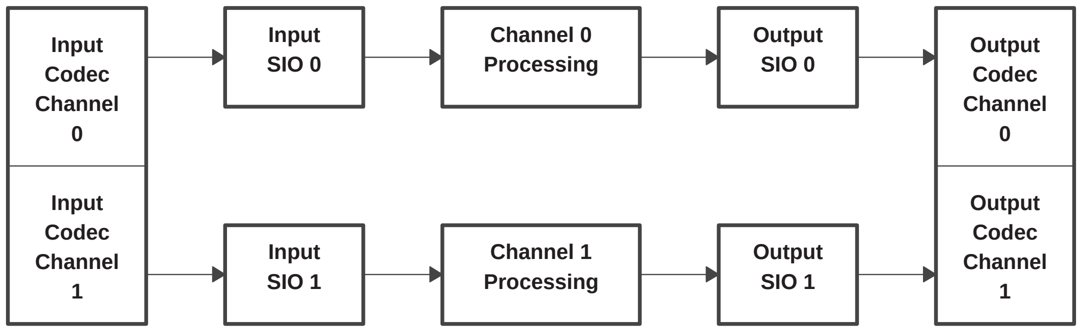

Figure 6. SIO with Single Task per Channel

SIO provides two models which can be used in this scenario: standard, which was discussed above, and issue/reclaim. The two approaches are discussed independently below. This architecture is ideal for systems with streams of differing rates or multiple algorithms. The priorities of the tasks can be set independently, which ensures that streams of different rates can all be handled within the real time deadlines.

2.2.1.1 Standard Model

Open the project in the directory /ti/myprojects/spra689/examples/*board_type*/examples/sio/sio3 (where *board_type* is dsk or evm). The echo routine is now simpler; it calls processChannel only once. This is because echo is the main routine of both channel tasks: echoTSK0 and echoTSK1. Each task knows which channel to process because the channel number is passed to the echo routine at task creation time. This is visible in the task properties of the individual tasks, in the DSP/BIOS II configuration. Currently, echoTSK0 immediately precedes the processing of echoTSK1. This can be changed by editing the task priorities of echoTSK1 in the DSP/BIOS II configuration. Change the priority to a higher number than the priority of echoTSK0, rebuild and reload, and run. Notice that channel 1 processing now precedes channel 0. See Figure 7.

By making each channel independent, we can assign different priorities to each one via the task properties. We also eliminate the overhead of checking which channel needs processing, or looping through all channels. On the execution graph, we can now see the processing for individual channels. As an experiment, display the audioChannel structure in the watch window. Edit the load field for channel 0 to be 500. Notice that the execution time for echoTSK0 increases, while for echoTSK1 it remains unchanged. See Figure 7.

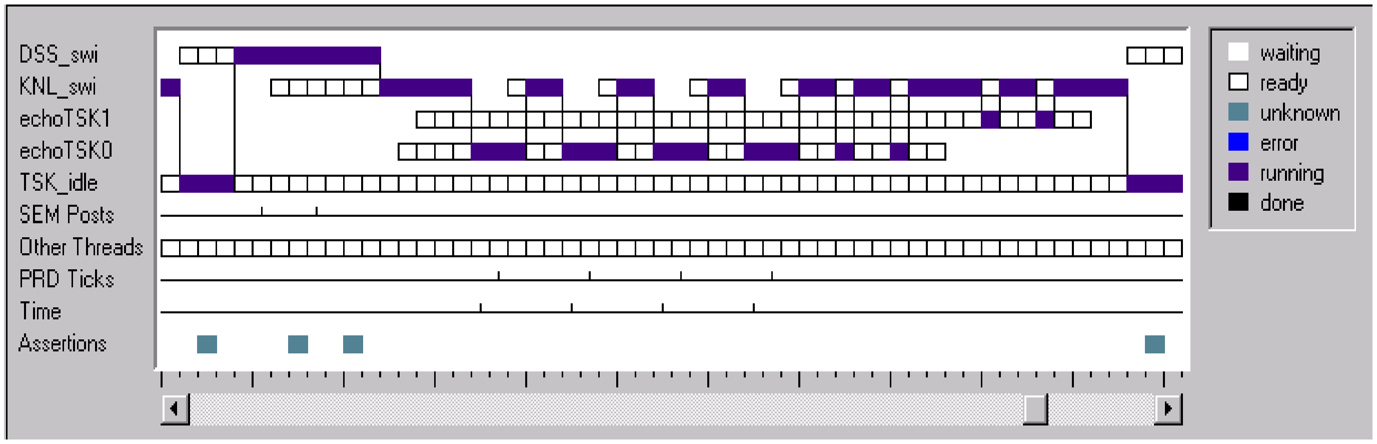

Figure 7. Execution Graph with Different Loads Per Channel

Each task services two SIOs: one for input and one for output. The routine processChannel is unchanged from the previous example. However, in initChannelData, a priming buffer is created for each channel. Previously, we only needed to create one priming buffer, because a single task serviced all the SIOs in the system. Once we separate the channels into separate tasks, each task will only service its input SIO and its output SIO; therefore, each task needs its own priming buffer.

Note that initChannelData is called from main; this is because echoTSK0 and echoTSK1 are created after main exits, and each requires the audioChannel structure to be initialized at startup. An alternative would have been to have each task perform the initialization of its own data structure.

This is a simple example, which has each channel performing identical processing. But if the channels contained different data which required different algorithms, the change would be easy; modify the routine name in the task properties in the DSP/BIOS II configuration, and provide the new routine, along with its necessary initializations.

2.2.1.2 Issue/Reclaim Model

Open the project in the /ti/myprojects/spra689/examples/*board_type*/examples/sio/sio4 directory (where *board_type* is dsk or evm). The issue/reclaim model of SIO differs from the standard model in that much of the buffer management performed by the standard model must be done by the application which uses issue/reclaim. The issue/reclaim model is designed to give the designer more control over the buffer management; the price for more control is more responsibility. In addition, the issue/reclaim model provides non-blocking I/O by combining SIO_issue, which does not block, with SIO_reclaim, which specifies a timeout value.

The issue/reclaim model does not automatically create the necessary buffers; nor does it perform a buffer exchange with the stream.

The buffer allocation may be done via MEM_alloc, or by checking the Allocate Static Buffers box in the SIO properties. This example was designed to be functionally equivalent to the standard model implementation in the previous example. Therefore, each stream is declared to have a capacity of three buffers. In the standard model, the stream has the capacity to hold the number of buffers declared plus one. At initialization time, the stream will have the two buffers that were declared. Then, when the first SIO_get is executed, a third buffer is passed in to the stream; hence the need for one extra. To mirror this functionality with the issue/reclaim model, we must declare three buffers because in the issue/reclaim model, the stream has the capacity for exactly as many buffers as are declared.

The basic components of the issue/reclaim model of SIO are the SIO_issue and SIO_reclaim API calls. SIO_issue gives a buffer to an SIO. SIO_reclaim gets an available buffer from an SIO. If the stream is defined as input, the buffer given by SIO_issue is treated as empty, to be filled by the device connected to the SIO. In this case, SIO_reclaim returns a buffer which has been filled by the device. If the SIO is declared as output, the reverse is true. Therefore, in the issue/reclaim model, an operation on an SIO requires a pair of operations: SIO issue to give the stream a buffer, and SIO_reclaim to get it back. Additionally, SIO_reclaim will return buffers in the same order in which they were given to the stream with SIO_issue. SIO_reclaim will cause the calling task to pend if there is no available buffer. A timeout may be specified for the reclaim operation.

The SIO standard model is actually built on the issue/reclaim model. A call to SIO_get or SIO_put generates a call first to SIO_issue, and then SIO_reclaim. This establishes the buffer exchange which is characteristic of the standard model. In the issue/reclaim model, this exchange must be done explicitly by the application.

Notice that the input SIOs have the Allocate Static Buffers box checked, whereas the output SIOs do not. As a result, at the start of the echo routine, SIO_staticbuf is called to get the addresses of the three input SIO buffers, while two buffers are MEM-alloc'ed for the output SIOs. Why only two buffers for the output SIO? Because in order to achieve double-buffering, only two buffers are needed. The reason the stream needs to hold three is for the period between the issue and the reclaim, when in fact the stream does contain three buffers. But that is internal to the SIO; the designer need only to supply it with two. In fact, this is also the case with the input SIOs. The reason for getting the address of the third buffer from the input SIO is to achieve the buffer switch required for double-buffering on input. The third buffer is used on the input SIO to mirror the buffer passed in with the first SIO_get in the standard model. All SIO_staticbuf calls for a particular SIO must be done before the first call to SIO_issue. MEM_alloc is used for the output buffers because, had we used statically-allocated buffers, one of those would be wasted memory, since three would be allocated.

Note that the first SIO_issue calls to the output SIO pass zero-length buffers. See Figure 8. This is required when priming an output SIO in issue/reclaim mode, if double-buffering is desired. Any buffer of non-zero length given to an output SIO will be transmitted. When priming the output SIO, we do not want to transmit anything; we only want the buffer to immediately be made available to the application when it calls SIO_reclaim, as if it had just been transmitted. This special case, of a zero-length buffer issued to an output SIO, must be handled by the underlying device driver. If the output SIO is not primed, then double-buffering will never take place on the output SIO. If your application is such that you do not need double-buffering, then you do not need to prime the output; but if you need it, the zero-length issue is the best approach.

```
/* give both buffers to outStream to prime it */
if (SIO_issue(audioChannel[channelNum].outStream,
                    audioChannel[channelNum].buf,
                    0,
                    NULL) < 0) {
      SYS_abort(”error on issue for chan %d”, channelNum);
}
if (SIO_issue(audioChannel[channelNum].outStream,
                    audioChannel[channelNum].buf1,
                    0,
                    NULL) < 0) {
      SYS_abort(”error on issue for chan %d”, channelNum);
}
```

Figure 8. Priming of Output SIO in Example sio4

To summarize the above information, in issue/reclaim mode:

1. Declare each SIO with one buffer more than is actually needed.
2. For output SIOs, MEM_alloc the buffers you need, and prime the SIO by issuing those buffers with length of zero.
3. For input SIOs, statically define all buffers, and issue all buffers to the SIO before doing the first reclaim.

processChannel has changed significantly from the previous examples. It first performs an SIO_issue on the input channel, this time with the other buffer. Note that the actual address of the buffer is passed to SIO_issue, NOT a pointer to a pointer, as was done with SIO_get. Now the input stream has two buffers with which to work; when the first is filled, it will start filling the second.

```
if (SIO_issue(channel–>inStream, channel–>buf, channel–>bufSize, NULL) < 0) {
    return(FALSE);
}
if (SIO_reclaim(channel–>inStream, &channel–>buf, &arg) < 0) {
    return(FALSE);
}
```

Figure 9. Processing Input Stream in Example sio4

SIO_issue does not pend, so immediately, SIO_reclaim is called, to get the previous buffer from the stream. See Figure 9. Note that in this call, a pointer to the buffer pointer is passed, since SIO_reclaim gives a buffer back to the application. If a buffer has not yet been filled, the task will pend within SIO_reclaim until the buffer is available. When SIO_reclaim exits, the previous buffer address has been returned to the task, while the current buffer is filling with data.

The buffer of input data is then processed, same as before. Next the task calls SIO_issue for the output stream, passing the address of the full buffer. Then SIO_reclaim is called for the output stream, which will pend until an empty buffer is available. The next time processChannel is called, the buffer passed in to SIO_issue will be the empty one previously returned from the SIO_reclaim on the output stream. Thus, the application manages the ping-pong of the buffers. The SIO module handles the low level buffer management as well as the interface with the device driver.

3 PIP

PIPs, also known as data pipes, offer an alternative approach to SIO for handling streaming data. The basic mechanism behind PIPs is very different from SIO. All PIPs and their buffers are statically created, as part of the PIP definition in the DSP/BIOS II configuration. PIPs may not be created or deleted dynamically, as SIOs can.

A PIP has a reader thread and a writer thread. The writer thread puts data into the buffers of the PIP; the reader thread extracts data from the buffers of the PIP. Reader and writer threads may be either HWIs, SWIs, or TSKs.

PIPs post SWIs when buffers become available. PIPs use a callback mechanism, also known as *notify functions*, to alert readers and writers of important PIP events. Readers are notified that a new buffer has arrived in the PIP, so it may be read. Writers are notified that an empty buffer has been returned to the PIP, and may be written. Whereas SIOs use the underlying DEV device driver interface to the hardware, PIPs do not. The structure of the PIP driver is left to the designer.

The four API calls that are necessary to read from and write to PIPs are: PIP_alloc, PIP_put, PIP_get, and PIP_free.

PIP_alloc and PIP_put are called by the writer thread of a PIP. PIP_get and PIP_free are called by the reader thread. The order of the calls is very important, and is described in Figure 10.

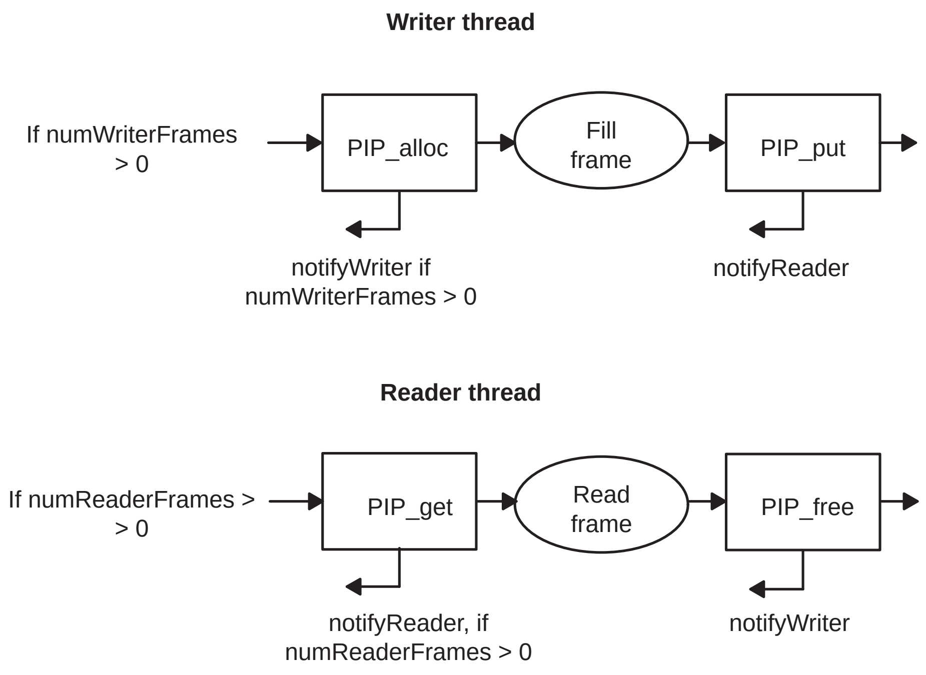

Figure 10. PIP API Calls and Callbacks

The writer thread must first check that the number of writer frames in the PIP is at least 1. A frame equates to a buffer. Writer frames are empty frames, which may be filled. If there is an available writer frame, PIP_alloc is called. PIP_alloc readies an available writer frame for writing by the application. After the frame is allocated, if the number of writer frames is still greater than 0, the notify Writer function is called from within PIP_alloc. This is done to let the writer know there is a frame available which he may write. The application then accesses the frame, and fills it with data. It then calls PIP_put, to put the filled frame into the PIP. This causes the notifyReader function to execute, to let the reader know that there is a new frame full of data to read.

The reader thread must first check that the number of reader frames in the PIP is at least 1. Reader frames are full frames. If there is an available reader frame, PIP_get is called. PIP_get readies an available reader frame for reading by the application. After this is done, if the number of reader frames is still greater than 0, the notifyReader function is called from within PIP_get. This is done to let the reader know there is a frame available which may be read. The application then accesses the frame, and reads the data. It then calls PIP_free, to return the empty frame into the PIP. This causes the notifyWriter function to execute, to let the writer know that there is a new empty frame available for writing.

Additional APIs are supplied for obtaining frame address, size, etc, for both reader and writer. But this data may be obtained directly from the PIP structure to save time. Accessing the PIP structure is recommended only for reading. Writing directly to the PIP structure may have undefined results.

For writers and readers, the calling order shown in the figure must be maintained: PIP_alloc followed by PIP_put, and PIP_get followed by PIP_free.

Whereas SIO requires the TSK threading model, PIP may be used with the SWI model (software interrupt) as well as the TSK model. SWIs differ from TSKs in several ways. SWIs are patterned after HWIs; that is, they are triggered by an event. In the case of SWI, the event is a call to an SWI function by the application. Whereas TSKs usually consists of an infinite loop which pends waiting for some system resource and resumes execution when the resource becomes available, SWIs are routines with an entry and an exit that may be blocked by HWIs or higher priority SWIs, but do not pend. SWIs exist in the system only once they have been posted, and until they exit, while TSKs exist until they are deleted or they exit. Also, SWIs share the system stack for their duration, while TSKs have their own local stack. Therefore, any solution using SWIs will use less memory for stack usage than a solution using TSKs.

The difference that is most important in the context of this note is that TSKs pend waiting for a resource to become ready, while SWIs are posted only when the necessary resources have become ready. Therefore, SWIs assume the availability of their needed resources; it is the responsibility of the designer to ensure the SWI is not posted until those resources are ready.

3.1 Using a Single SWI Per Channel

Open the project in the /ti/myprojects/spra689/exmples/*board_type*/examples/pip/pip1 directory (where *board_type* is dsk or evm). In the DSP/BIOS II configuration, notice one receive-PIP and one transmit-PIP per channel. These are for input and output of data, respectively. procSWI0 and procSWI1 are the threads that actually read the input data from the receive PIP, process it, and write it to the transmit PIP for output.

Figure 11 diagrams the data and control flow of this code example. This diagram encompasses all PIP calls and callbacks made in the example, in an attempt to tie together all the elements of a PIP-based design. Channel 0 elements and flow are in the top half; channel 1 is in the bottom half.

The data flow is depicted by the thick solid arrows. There’s nothing surprising here; the data comes in via the codec, goes into an rxPip, is processed by a procSwi, is transferred to a txPip, and output via codec. The really interesting story is in the control flows, which are depicted by thin solid arrows. All notifyWriter and notifyReader callbacks are represented here, along with all PIP API calls in the example.

Although the device driver details are beyond the scope of this note, the parts of the driver that use the PIP API will be mentioned, because they are important to the understanding of how the example works. For the details of the device driver, see SPRA700.

At system initialization, all notifyWriter functions execute, because all PIPs contain writable (empty) frames. The driver starts the stream by doing a PIP_alloc from the rxPips, which gets a buffer form the PIP to fill with incoming data. Once the buffer is filled, the driver does a PIP_put, putting the filled frame into the rxPip. This triggers the notifyReader routine. The reader for the rxPip is the procSwi. **Note that neither procSWI0 nor procSWI1 will execute until it has been notified that a reader frame is available AND a writer frame is available.** This is accomplished by the callbacks for the PIPs calling SWI_andn. rxPips use this routine to set one bit; txPips use it to set a different bit. Once both bits are set, procSwi will run. This reflects the SWI model philosophy of not running until all resources are available.

The procSwi does a PIP_get from the rxPip to get the input frame. It processes the frame data, and does a PIP_alloc from the txPip to get an empty frame for output. The data from the input buffer is transferred to the output buffer, and then procSwi calls PIP_free to return the input frame to the rxPip, since it has completed processing on that frame. Calling PIP_free causes the rxPip notifyWriter routine to execute because the writer (the driver) needs to know that there a frame is now available on the rxPip.

The driver then readies a new rxPip frame for use. procSwi, having allocated a frame from the txPip and filled it with the data to output, now calls PIP_put to give the filled frame to the txPip. This causes the notifyReader function of txPip to execute, so that the reader thread will know there is now a full frame ready to be read. The reader of the txPips is the driver, which then calls PIP_get to retrieve the full frame from the txPip. The driver sets up the transmit hardware to send the frame. When the frame transmit has completed, the driver calls PIP_free, to free the now-empty frame back to the txPip. The txPip notifyWriter routine is then called to notify procSwi, which is the writer for txPip, that a frame is now available for writing. It does this by calling SWI_andn with a specified bit.

With this bit set, half of procSwi’s requirement to run has been fulfilled. The next time a receive buffer arrives, this cycle will repeat.

The important aspects of this example are:

- The driver is the writer for both rxPips, and the reader for both txPips
- The procSwis are readers for the rxPips, and writers for txPips
- The procSwis are only run after both a full buffer is ready for input AND an empty buffer is ready for output

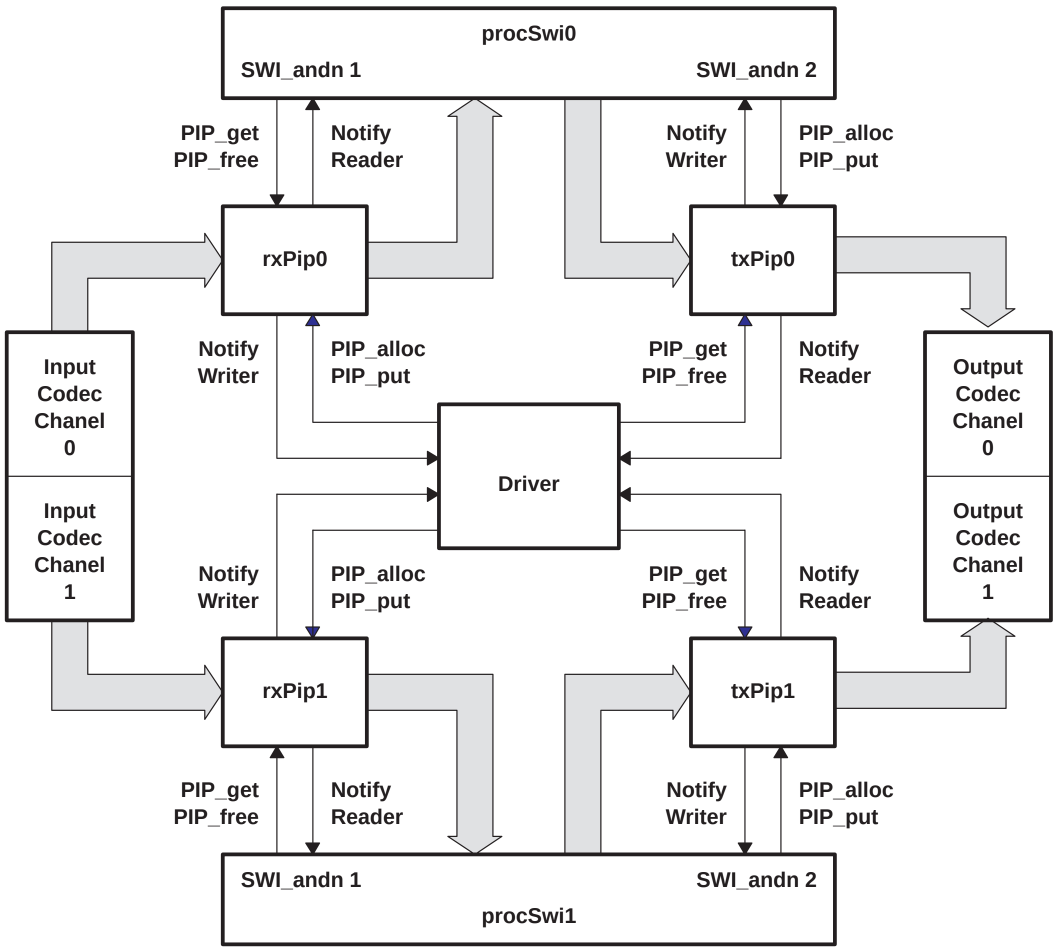

Figure 11. Data and Control Flow: Single SWI PIP Example

Note that the driver initialization function (DSS_init for the EVM example and IPI_LIO_init for the DSK example) is called explicitly from main, whereas in the SIO examples this was not the case. This is because SIO uses the DEV functions to do all device-specific operations. In the SIO case, the driver initialization function is called as part of SIO_init, which is handled by DSP/BIOS II as part of system startup. PIP does not use the DEV device driver model; it has no mechanism to automatically initialize the device driver. PIP must do this initialization explicitly, so driver initialization function must be called by the application. You will also need to call your DEV_init function explicitly if your application uses ONLY dynamically created SIOs. Although the driver initialization function enables interrupts for the driver, it does not attempt to enable interrupts globally, so its placement in main is acceptable (interrupts are not enabled globally until main exits).

3.1.1 *notifyReader and notifyWriter Routines*

When the reader or writer for a PIP is a device driver, the callback routines used by the PIP (also known as *notifyReader* and *notifyWriter*), are sometimes a source of confusion. This is partly because they tend to involve some driver functionality, and partly because many existing code examples use the same routines for the callbacks as they do in the driver ISR. The result is several if-statements, usually checking if the routine has been called by an ISR, so special processing can take place. This section separates the functionality that needs to be in the callback from the functionality that needs to go into the ISR (apart from normal ISR processing, of course; that is beyond the scope of this note).

This section presents PIP-specific pseudo-code for the callback routines, as well as PIP-specific pseudo-code for ISR functionality, when the reader or writer is a device driver.

3.1.1.1 Device Driver as Writer

A device driver will be the writer to a PIP on the input, or “receive”, end of a system. In Figure 11, although the data path goes directly from input codec to rxPip, technically this is not quite the case. The data comes from the codec to the driver, which then writes the data to the rxPip. These data paths were omitted because for the purposes of the diagrams, the driver and the hardware represent the same thing; and also to make the diagrams more readable.

Therefore, the notifyWriter callback must “prime” the system for a new buffer. In this context, “prime” means to set up a new buffer address and receive count for the hardware. Therefore, the pseudo-code for the notifyWriter functionality is as follows:

```
ReceivePrimeCallback()
{
   If (receive is not primed) {
      Get new buffer (frame) from PIP
      Set up new buffer address and count
   }
}
```

Figure 12. notifyWriter Pseudo-code for Receive

The specifics of implementation may vary, but the goal in this routine is to get the hardware started, if is not already, to receive more data into the newly-free’d frame.

The ISR functionality to support PIPs is:

```
ReceiveISR()
{
   /* perform other necessary ISR functions for receive */
   if (receiveBuffer is full) {
      place the buffer (frame) back in the PIP
      get a new frame from PIP
      if (no frame available) {
         clear buffer address and count (frame-drop processing)
      }
      else {
         set up new buffer address and count
      }
   }
}
```

Figure 13. ISR Pseudo-code for Receive

This functionality feeds the PIP with a new buffer after it has been completely received by the hardware. Depending on the specifics of your application, the content of your ISR will vary (eg., DMA vs McBSP), but the overall functionality must reflect these operations .

3.1.1.2 Device Driver as Reader

A device driver will be the reader of a PIP on the output, or “transmit”, end of a system. In Figure 11, although the data path goes directly from txPip to output codec, technically this is not quite the case. The data goes from the txPip to the driver, which then writes the data to the codec. These data paths were omitted because for the purposes of the diagrams, the driver and the hardware represent the same thing; and also to make the diagrams more readable.

Therefore, the notifyReader callback must “prime” the system to output a new buffer. In this context, “prime” means to set up a new buffer address and send count for the hardware. Therefore, the pseudo-code for the notifyReader functionality is as follows:

```
TransmitPrimeCallback()
{
   If (transmit is not primed) {
      Get new buffer (frame) from PIP
      Set up new buffer address and count
   }
}
```

Figure 14. notifyReader Pseudo-code for Transmit

The specifics of implementation may vary, but the goal in this routine is to get the hardware started, if is not already, to send more data from the newly-filled frame.

The ISR functionality to support PIPs is:

```
TransmitISR()
{
   /* perform other necessary ISR functions for transmit */
   if (transmitBuffer is empty) {
      place the buffer (frame) back in the PIP
      get a new frame from PIP
      if (no frame available) {
         clear buffer address and count (frame-drop processing)
      }
      else {
         set up new buffer address and count
      }
   }
}
```

Figure 15. ISR Pseudo-code for Transmit

This functionality feeds the hardware (driver) with a new buffer after it has been filled with data by the application. Depending on the specifics of your application, the content of your ISR will vary (eg., DMA vs McBSP), but the overall functionality must reflect these operations.

4 Summary

On the surface, PIP and SIO appear to be very different; but they are just two different ways of accomplishing the same thing: managing buffers for device-independent I/O in real time. Both have a single reader thread and a single writer thread.

PIP uses less memory and is faster, whereas SIO is easier to use and more flexible. PIP is a non-blocking mechanism, whereas SIO may be configured to block or not. SIO may only be used with the TSK threading model, whereas PIP may be used with both TSK and SWI. Each PIP owns its buffers; the memory cannot be reclaimed. SIOs may statically create the buffers, or they may be created by other means; buffers created statically in one SIO may be transferred to another. SIO may be created and deleted statically via the DSP/BIOS configuration tool, or dynamically via API; PIP can only be created statically, and is never deleted.

5 References

1. *An Audio Example Using DSP/BIOS II* (SPRA598)
2. *TMS320C6000 DSP/BIOS II User's Guide* (SPRU303)
3. *Benchmarking DSP/BIOS II on the C600*0 (SPRA662)
4. *Writing Flexible Device Drivers for DSP/BIOS* (SPRA700)