DSP II: ELEC 4523

SIO Module

Objectives

- Become familiar with SIO objects and their use with the DPI driver of the DEV module.

Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Streaming I/O and Device Drivers (chapter)
- PowerPoint Slides from DSP/BIOS II Workshop: Module 8
- Code Composer Studio Online Help: SIO Module, DPI Driver

Lab Module Prerequisites

Be sure to do the TSK and QUE modules before the SIO module.

Description

Pipes and streams are both used to transfer data from one task to another. Both methods must have a single reader thread and a single writer thread. Also, both methods transfer data by copying pointers to buffers rather than copying the buffers. There are a few differences between the two methods as seen in Table 1. Pipes generally support low-level communication while streams support high-level, device independent communication.

The SIO module is a very flexible module that allows interfacing to many different types of devices as well as user defined devices. This lab assignment focuses on a very narrow portion of the SIO module and its use. For more detailed information you should consult the reading above as well as SPRU403 TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide.

DSP applications usually receive data from a device, process the data and output the results to a device. This processing happens continually. The SIO module is a general interface for application software to a particular device. Function calls are the same no matter what type of device the data is comming from or going to. The DEV module is designed to interface to different types of devices. The SIO function calls, which are general I/O calls, are associated with DEV function calls which are particular for the device that is being used.

Pipes (PIP and HST)	Streams (SIO and DEV)
Programmer must create own driver structure.	Provides a more structured approach to device-driver creation.
Reader and writer can be any thread type or host PC.	One end must be handled by a task (TSK) using SIO calls. The other end must be handled by an HWI using Dxx calls.
PIP functions are non-blocking. Program must check to make sure a buffer is available before reading from or writing to the pipe.	SIO_put, SIO_get, and SIO_reclaim are blocking functions and causes a task to wait until a buffer is available. (SIO_issue is non-blocking.)
Uses less memory and is generally faster.	More flexible; generally simpler to use.
Each pipe owns its own buffers.	Buffers can be transferred from one stream to another without copying. (In practice, copying is usually necessary anyway because the data is processed.)
Pipes must be created statically with the Configuration Tool.	Streams can be created either at run time or statically with the Configuration Tool. Streams can be opened by name.
No built-in support for stacking devices.	Support is provided for stacking devices.
Using the HST module with pipes is an easy way to handle data transfer between the host and target.	A number of device drivers are provided with DSP/BIOS.

Table 1: Comparison of Pipes and Streams, taken from *SPRU423 TMS320 DSP/BIOS Users Guide*

There are three different basic configurations for stream devices as shown in Figure 1. Task communication is the configuration used in this laboratory. In task communication there are two tasks that need to transmit data from one to the other. In between the two tasks are two streams and a device driver specifcally for transfering data between tasks. This device driver is called the DPI driver, or the pipe driver. Figure 2 shows a general structure for communication between two tasks.

Streams implement I/O by copying pointers. Streams have a queue of free buffers and a queue of full buffers. Streams can be set up as either input or output. If a stream is set up for input then a full buffer is supplied by the program and swapped for a free buffer. The stream keeps the full buffer and returns a free buffer. If a stream is set up for output then a free buffer is supplied by the program and swapped for a full buffer. The stream keeps the free buffer and returns a full buffer.

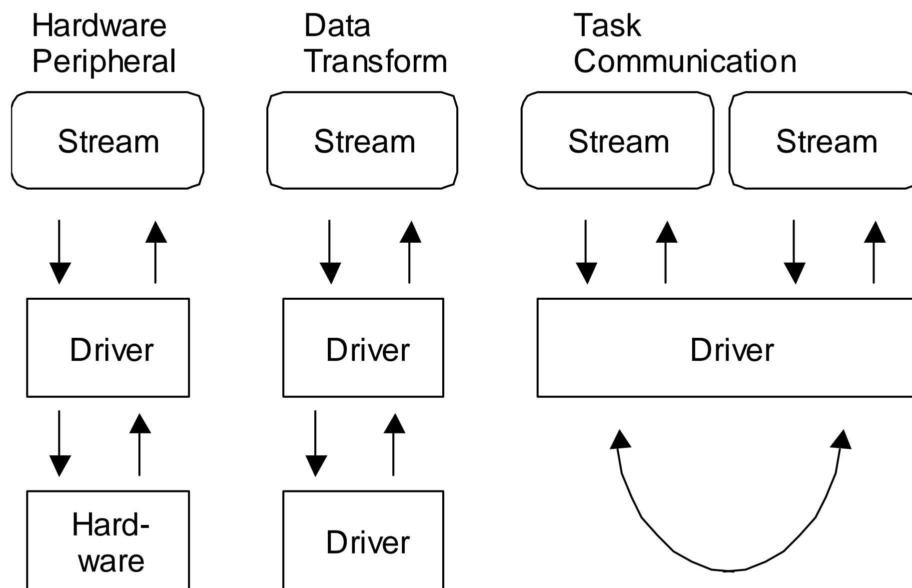

Figure 1: Types of stream devices, taken from *PowerPoint Slides from DSP/BIOS II Workshop: Module 9*

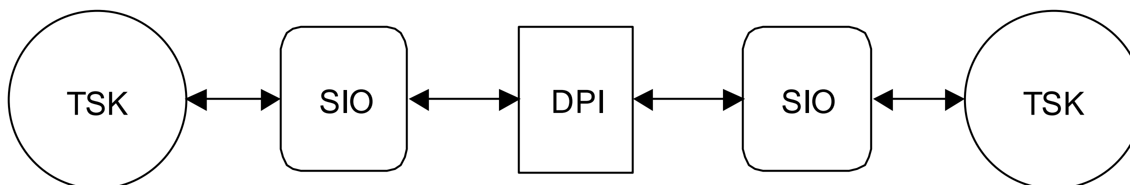

Figure 2: Task to task communication with streams

This laboratory uses streams in the standard mode, `SIO_STANDARD`. A few of the main functions for using streams in this mode are:

- `SIO_bufsize` - Size of the buffers used by a stream
- `SIO_segid` - Memory segment used by a stream
- `SIO_put` - Put buffer to a stream
- `SIO_get` - Get buffer from stream
- `SIO_staticbuf` - Acquire static buffer from stream

When an SIO object is initialized it can be initialized with buffers. The buffer size, memory segment and number of buffers are parameters to the SIO object. When exchanging buffers with an SIO object the buffer being exchanged must have the same size and be in the same memory segment as the SIO object buffers.

The function `SIO_get` is used to receive a full buffer from the stream. First an empty buffer must be obtained and then passed to `SIO_get` which exchanges the pointers. If static buffers have been initialized then the function `SIO_staticbuf` can be used to get a buffer from the stream. The following code gets an empty buffer from the input stream and then gets a full buffer from the stream by exchanging the buffers.

```
SIO_Handle input = &inputStream;
Int *buf;
/* get an empty buffer from the stream */
if ( SIO_staticbuf(input, (Ptr *)&buf) == 0) {
    SYS_abort("Error reading buffer ");
}
/* get a full buffer from the stream by exchanging pointers */
if ((nbytes = SIO_get(input, (Ptr *)&buf)) < 0) {
    SYS_abort("Error reading buffer %d", i);
}
```

To put a full buffer in the stream the following code is used:

```
SIO_Handle output = &outputStream;
Int *buf;
/* get the number of bytes in the stream buffers */
nbytes = SIO_bufsize(output);
/* get a static empty buffer from the stream */
if ( SIO_staticbuf(output, (Ptr *)&buf) == 0) {
    SYS_abort("Error reading buffer ");
}
if ((nbytes = SIO_put(output, (Ptr *)&buf,nbytes)) < 0) {
    SYS_abort("Error putting buffer");
}
```

Laboratory

Part 1

- In this part you will be creating two TSKs were one will generate some data and send it to the other TSK in a stream. The transmitting TSK will generate only 3 buffers of data to send to the other TSK. The data stream will be initialized with two buffers.
- Create a new project called `siolab`.
- Create a new DSP/BIOS Configuration file and use the `C6xxx.cdb` template for use with the simulator.
- Save the file as `siolab.cdb` and add it in your project. Also add the `siolabcfg.cmd` file.
- If using the simulator then change the RTDX interface to Simulator by right clicking on Input/Output->RTDX and bringing up the properties. Change the RTDX mode to Simulator. If you do not do this then when you load your program you will see the error "RTDX application does not match emulation protocol." If you are loading onto an EVM or DSK you shouldn't need to change this.
- Create a LOG object by right clicking on Instrumentation->LOG and selecting Insert LOG. Change the name to `trace`. Set its properties to have a length of 512 and be a circular buffer.
- Change the `LOG_system` object to have a length of 512.
- Create a DPI driver by right clicking on Input/Output->SIO->SIO Drivers->DPI and selecting Insert DPI. The new driver will be called `DPI0`.
- Create two streams with the following properties:

  - Name: `inputStream`, Device: DPI0, Buffer size: 128, Number of Buffers: 2, Mode: input, check Allocate Static Buffers.
  - Name: `outputStream`, Device: DPI0, Buffer size: 128, Number of Buffers: 2, Mode: output, check Allocate Static Buffers.
- Create two TSKs with the following properties
  - Name: `TSK0`, priority: 1, function: `_funTSK0`.
  - Name: `TSK1`, priority: 1, function: `_funTSK1`.
- Create a main.c file and include a `main` function that prints to `trace` a message saying the run is starting. Include this file in your project.
- In the main.c file make functions for your TSKs, `funTSK0` and `funTSK1`.
- In funTSK0 add the following code
  - Get the size of the buffer in `inputStream`.
  - Get a free buffer from `inputStream`.
  - Make a loop that loops 3 times.
  - In the loop fill the buffer with numbers from 0 to the end of the buffer. Use `sizeof(Int)` to get the number of bytes per Int to determine the number of times to loop.
  - In the loop put the buffer in the `inputStream`.
- In funTSK1 add the following code
  - Get a free buffer from `outputStream`.
  - Make a loop that loops 3 times.
  - In the loop get a full buffer from `outputStream`. Use `SIO_get` to get the buffer and the number of bytes in the buffer.
  - In the loop print the contents of the buffer and the task doing the printing.
- Build and load your project.
- Use the Execution Graph and the LOG manager to examine the processing. Open the RTA Control Plannel by selecting it under the DSP/BIOS menu. Uncheck the Enable SWI logging. This will cause the KNL_swi to not be logged since we are not interested in when it runs.
- Run the program and record the results.
- Describe in detail the processing that is occurring.

Part 2

- Change the priority of `TSK1` to 2.
- Build and load your project.
- Run the program and record the results.
- How does the execution change? Describe in detail.