ADI_SPI_DMA DEVICE DRIVER

DATE: 01 APRIL 2008

Table of Contents

Table of Figures

Document Revision History

Date	Description of Changes
12 Feb 2006	Document created
23 Oct 2006	Added Operation section
31 Oct 2006	Updated event code section
01 Apr 2008	Corrected example code; initialization of pNext field of last entry in buffer list
05 Jan 2010	Updated with command to set SPI Pin Mux

Table 1 - Revision History

1. Overview

The SPI Device Driver is designed to provide a simple interface with Blackfin SPI Peripherals. The commands, events and return codes in device driver can be used by the application programs to establish communication with the SPI. The SPI device driver supports both DMA-driven and interrupts driven versions.
Two source files adi_spi_dma.c and adi_spi_int.c are provided to select DMA-driven or interrupt driven version of the SPI device driver.

This document only describes the DMA driven SPI driver.
To select DMA-driven version of the SPI device driver, the application code shall

- Include adi_spi_dma.c in the project files.
- Use DMA entry point “ADISPIDmaEntryPoint” to open the SPI device driver.

2. Files

The files listed below comprise the device driver API and source files.

2.1. Include Files

The driver sources include the following include files:

- <services/services.h> This file contains all definitions, function prototypes etc. for all the System Services.
- <drivers/adi_dev.h> This file contains all definitions, function prototypes etc. for the Device Manager and general device driver information.
- <drivers/spi/adi_spi.h> This file contains all definitions, function prototypes etc. for the SPI driver.

2.2. Source Files

The driver sources are contained in the following files, as located in the default installation directory:

- adi_spi.c
- adi_spi_dma.c

3. Lower Level Drivers

The SPI driver does not use any lower level driver.

4. Resources Required

Device drivers typically consume some amount of system resources. This section describes the resources required by the device driver.

Unless explicitly noted in the sections below, this device driver uses the System Services to access and control any required hardware. The information in this section may be helpful in determining the resources this driver requires, such as the number of interrupt handlers or number of DMA channels etc., from the System Services.

Because dynamic memory allocations are not used in the Device Drivers or System Services, all memory used by the Device Drivers and System Services must be supplied by the application. The Device Drivers and System Services supply macros that can be used by the application to size the amount of base memory and/or the amount of incremental memory required to support the needed functionality. Memory for the Device Manager and System Services is provided in the initialization functions (adi_xxx_Init()).

4.1. Interrupts

This driver does not use interrupt.

4.2. DMA

This driver uses one DMA channel which can be configured to support either an SPI transmit channel or a receive channel.

4.3. Timers

This driver does not require the timers.

4.4. Real-Time Clock

This driver does not require the real-time clock.

4.5. Programmable Flags

This driver does not require the programmable flags.

4.6. Pins

This driver uses two data pins MOSI and MISO, one device select pin SPISS and a clock pin CLK. Please check Blackfin hardware reference manual for more details.

5. Supported Features of the Device Driver

This section describes what features are supported by the device driver.

5.1. Directionality

The driver supports the dataflow directions listed in the table below.

ADI_DEV_DIRECTION	Description
ADI_DEV_DIRECTION_INBOUND	Supports the reception of data in through the device.
ADI_DEV_ DIRECTION_OUTBOUND	Supports the transmission of data out through the device.

Table 2 - Supported Dataflow Directions

5.2. Dataflow Methods

The driver supports the dataflow methods listed in the table below.

ADI_DEV_MODE	Description
ADI_DEV_MODE_CHAINED	Supports the chained buffer method
ADI_DEV_MODE_CHAINED_LOOPBACK	Supports the chained buffer with loopback method

Table 3 - Supported Dataflow Methods

5.3. Buffer Types

The driver supports the buffer types listed in the table below.

- ADI_DEV_1D_BUFFER
  - Linear one-dimensional buffer
  - pAdditionalInfo – ignored

5.4. Command IDs

This section enumerates the commands that are supported by the driver. The commands are divided into three sections. The first section describes commands that are supported directly by the Device Manager. The next section describes common commands that the driver supports. The remaining section describes driver specific commands.

Commands are sent to the device driver via the adi_dev_Control() function. The adi_dev_Control() function accepts three arguments:

- DeviceHandle – This parameter is a ADI_DEV_DEVICE_HANDLE type that uniquely identifies the device driver. This handle is provided to the client in the adi_dev_Open() function call.
- CommandID – This parameter is a u32 data type that specifies the command ID.
- Value – This parameter is a void * whose value is context sensitive to the specific command ID.

The sections below enumerate the command IDs that are supported by the driver and the meaning of the Value parameter for each command ID.

5.4.1. Device Manager Commands

The commands listed below are supported and processed directly by the Device Manager. As such, all device drivers support these commands.

- ADI_DEV_CMD_TABLE
    - Table of command pairs being passed to the driver
    - Value – ADI_DEV_CMD_VALUE_PAIR *
- ADI_DEV_CMD_END
    - Signifies the end of a command pair table
    - Value – ignored
- ADI_DEV_CMD_PAIR
    - Single command pair being passed
    - Value – ADI_DEV_CMD_PAIR *
- ADI_DEV_CMD_SET_SYNCHRONOUS
    - Enables/disables synchronous mode for the driver
    - Value – TRUE/FALSE

5.4.2. Common Commands

The command IDs described in this section are common to many device drivers. The list below enumerates all common command IDs that are supported by this device driver.

- ADI_DEV_CMD_GET_2D_SUPPORT
    - Determines if the driver can support 2D buffers
    - Value – u32 * (location where TRUE/FALSE is stored)
- ADI_DEV_CMD_SET_DATAFLOW_METHOD
    - Specifies the dataflow method the device is to use. The list of dataflow types supported by the device driver is specified in section 5.2.
    - Value – ADI_DEV_MODE enumeration
- ADI_DEV_CMD_SET_STREAMING
    - Enables/disables the streaming mode of the driver.
    - Value – TRUE/FALSE
- ADI_DEV_CMD_GET_INBOUND_DMA_CHANNEL_ID
    - Returns the DMA channel ID value for the device driver's inbound DMA channel
    - Value – u32 * (location where the channel ID is stored)
- ADI_DEV_CMD_GET_OUTBOUND_DMA_CHANNEL_ID
    - Returns the DMA channel ID value for the device driver's outbound DMA channel
    - Value – u32 * (location where the channel ID is stored)
- ADI_DEV_CMD_SET_INBOUND_DMA_CHANNEL_ID
    - Sets the DMA channel ID value for the device driver's inbound DMA channel
    - Value – ADI_DMA_CHANNEL_ID (DMA channel ID)
- ADI_DEV_CMD_SET_OUTBOUND_DMA_CHANNEL_ID
    - Sets the DMA channel ID value for the device driver's outbound DMA channel
    - Value – ADI_DMA_CHANNEL_ID (DMA channel ID)
- ADI_DEV_CMD_GET_INBOUND_DMA_PMAP_ID
    - Returns the PMAP ID for the device driver's inbound DMA channel
    - Value – u32 * (location where the PMAP value is stored)
- ADI_DEV_CMD_GET_OUTBOUND_DMA_PMAP_ID
    - Returns the PMAP ID for the device driver's outbound DMA channel
    - Value – u32 * (location where the PMAP value is stored)
- ADI_DEV_CMD_SET_DATAFLOW
    - Enables/disables dataflow through the device
    - Value – TRUE/FALSE
- ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT
    - Determines if the device driver is supported by peripheral DMA
    - Value – u32 * (location where TRUE or FALSE is stored)

- ADI_DEV_CMD_SET_ERROR_REPORTING
  - Enables/Disables error reporting from the device driver
  - Value – TRUE/FALSE

5.4.3. Device Driver Specific Commands

The command IDs listed below are supported and processed by the device driver. These command IDs are unique to this device driver.

- ADI_SPI_CMD_SET_BAUD_RATE
  - Set the baud rate
  - Value = frequency in Hz
- ADI_SPI_CMD_SET_SLAVE_FLAG
  - Set the SPI slave flag register
  - Value = register value
- ADI_SPI_CMD_SET_CONTROL_REG
  - Set the SPI control register directly
  - Value = register value
- ADI_SPI_CMD_SET_BAUD_REG
  - Set the baud register directly
  - Value = register value
- ADI_SPI_CMD_ENABLE_SLAVE_SELECT
  - Enables a slave select line
  - Value = slave number
- ADI_SPI_CMD_DISABLE_SLAVE_SELECT
  - Disables a slave select line
  - Value = slave number
- ADI_SPI_CMD_SELECT_SLAVE
  - Selects a slave
  - Value = slave number
- ADI_SPI_CMD_DESELECT_SLAVE
  - De-selects a slave
  - Value = slave number
- ADI_SPI_CMD_SET_EXCLUSIVE_ACCESS
  - Prevent others from using SPI
  - Value = TRUE - want access, FALSE - done with access
- ADI_SPI_CMD_SET_TRANSFER_INIT_MODE
  - Sets TIMOD value
  - Value = 0 - start with core read of SPI_RDBR (interrupt driven inbound data)
  - Value = 1 - start with core write of SPI_TDBR (interrupt driven outbound data)
  - Value = 2 - start with DMA read of SPI_RDBR (DMA driven inbound data)
  - Value = 3 - start with DMA write of SPI_TDBR (DMA driven outbound data)
- ADI_SPI_CMD_SEND_ZEROS
  - Sends zeros when TDBR is empty
  - Value =TRUE - sends zeros, FALSE - send last data
- ADI_SPI_CMD_SET_GET_MORE_DATA
  - Sets the get more data mode
  - Value = TRUE - gets more date (overwrite), FALSE - discard incoming data
- ADI_SPI_CMD_SET_PSSE
  - Sets the PSSE bit in control
  - Value = TRUE - SPISS enabled, FALSE - SPISS disabled
- ADI_SPI_CMD_SET_MISO
  - Enables/disables MISO
  - Value = TRUE - MISO enabled, FALSE - MISO disabled
- ADI_SPI_CMD_SET_WORD_SIZE
  - Sets the number of bits per word
  - Value = 8 or 16

- ADI_SPI_CMD_SET_LSB_FIRST
  - Sets the MSB/LSB order
  - Value = TRUE - LSB sent first, FALSE - MSB sent first
- ADI_SPI_CMD_SET_CLOCK_PHASE
  - Sets the transfer format
  - Value = TRUE - beginning toggle, FALSE - middle toggle
- ADI_SPI_CMD_SET_CLOCK_POLARITY
  - Sets clock polarity
  - Value = TRUE - active low, FALSE - active high
- ADI_SPI_CMD_SET_MASTER
  - Sets master/slave control
  - Value = TRUE - master, FALSE – slave
- ADI_SPI_CMD_SET_OPEN_DRAIN_MASTER
  - Controls the WOM bit in SPI_CTL
  - Value = TRUE - open drain, FALSE - normal
- ADI_SPI_CMD_EXECUTE_DUMMY_READ
  - Causes a dummy SPI read to occur
  - Value = 8 - 8 bit read, 16 - 16 bit read
- ADI_SPI_CMD_PAUSE_DATAFLOW
  - Pauses/resumes dataflow
  - Value =TRUE - pause, FALSE – resume
- ADI_SPI_CMD_SET_PIN_MUX_MODE
  - Sets processor specific pin mux mode
  - Value = Enumeration of type ADI_SPI_PIN_MUX_MODE

Processor Family	ADI_SPI_PIN_MUX_MODE Enumeration value	Comments
ADSP - BF50x (Moy)	ADI_SPI_PIN_MUX_MODE_0	SPI Pin Mux configuration mode 0 (default) For SPI 0 - PF15 as Slave Select 3 (SPI0 SSEL3) For SPI 1 - PG1 as Slave Select 2 (SPI1 SSEL2) PG0 as Slave Select 3 (SPI1 SSEL3)
	ADI_SPI_PIN_MUX_MODE_1	SPI Pin Mux configuration mode 1 For SPI 0 - PH0 as Slave Select 3 (SPI0 SSEL3) For SPI 1 - PG1 as Slave Select 2 (SPI1 SSEL2) PH1 as Slave Select 3 (SPI1 SSEL3)
	ADI_SPI_PIN_MUX_MODE_2	SPI Pin Mux configuration mode 2 For SPI 0 - PF15 as Slave Select 3 (SPI0 SSEL3) For SPI 1 - PH2 as Slave Select 2 (SPI1 SSEL2) PG0 as Slave Select 3 (SPI1 SSEL3)
	ADI_SPI_PIN_MUX_MODE_3	SPI Pin Mux configuration mode 3 For SPI 0 - PH0 as Slave Select 3 (SPI0 SSEL3) For SPI 1 - PH2 as Slave Select 2 (SPI1 SSEL2) PH1 as Slave Select 3 (SPI1 SSEL3)
Other Processors	Command Not supported	

5.5. Callback Events

This section enumerates the callback events the device driver is capable of generating. The events are divided into two sections. The first section describes events that are common to many device drivers. The next section describes driver specific event IDs. The callback function of the client should be prepared to process each of the events in these sections.

The callback function is of the type ADI_DCB_CALLBACK_FN. The callback function is passed three parameters. These parameters are:

- ClientHandle – This void * parameter is the value that is passed to the device driver as a parameter in the adi_dev_Open() function.
- EventID – This is a u32 data type that specifies the event ID.
- Value – This parameter is a void * whose value is context sensitive to the specific event ID.

The sections below enumerate the event IDs that the device driver can generate and the meaning of the Value parameter for each event ID.

5.5.1. Common Events

The events described in this section are common to many device drivers. The list below enumerates all common event IDs that are supported by this device driver.

- ADI_DEV_EVENT_BUFFER_PROCESSED
  - Notifies callback function that a chained or sequential I/O buffer has been processed by the device driver. This event is also used to notify that an entire circular buffer has been processed if the driver was directed to generate a callback upon completion of an entire circular buffer.
  - Value – For chained or sequential I/O dataflow methods, this value is the CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_Read(), adi_dev_Write() or adi_dev_SequentialIO() function. For the circular dataflow method, this value is the address of the buffer provided in the adi_dev_Read() or adi_dev_Write() function.

- ADI_DEV_EVENT_DMA_ERROR_INTERRUPT
  - Notifies the callback function that a DMA error occurred.
  - Value – Null.

5.5.2. Device Driver Specific Events

The events listed below are supported and processed by the device driver. These event IDs are unique to this device driver.

- ADI_SPI_EVENT_TRANSMISSION_ERROR
  - TXE condition detected.
  - Value – Not used
- ADI_SPI_EVENT_RECEIVE_ERROR
  - RBSY condition detected
  - Value – Not used
- ADI_SPI_EVENT_MODE_FAULT_ERROR
  - MODF condition detected
  - Value – Not used
- ADI_SPI_EVENT_TRANSMIT_COLLISION_ERROR
  - TXCOL condition detected
  - Value – Not used

- ADI_SPI_EVENT_ERROR_INTERRUPT (***NOTE: This event is no longer used and has been replaced by the individual error events described above.***)
    - SPI error generated
    - Value – Not used

5.6. Return Codes

All API functions of the device driver return status indicating either successful completion of the function or an indication that an error has occurred. This section enumerates the return codes that the device driver is capable of returning to the client. A return value of ADI_DEV_RESULT_SUCCESS indicates success, while any other value indicates an error or some other informative result. The value ADI_DEV_RESULT_SUCCESS is always equal to the value zero. All other return codes are a non-zero value.

The return codes are divided into two sections. The first section describes return codes that are common to many device drivers. The next section describes driver specific return codes. Wherever functions in the device driver API are called, the application should be prepared to process any of these return codes.

Typically the application should check the return code for ADI_DEV_RESULT_SUCCESS, taking appropriate corrective action if ADI_DEV_RESULT_SUCCESS is not returned. For example:

```
if (adi_dev_Xxxx(…) == ADI_DEV_RESULT_SUCCESS) {
      // normal processing
} else {
      // error processing
}
```

5.6.1. Common Return Codes

The return codes described in this section are common to many device drivers. The list below enumerates all common return codes that are supported by this device driver.

- ADI_DEV_RESULT_SUCCESS
    - The function executed successfully.
- ADI_DEV_RESULT_NOT_SUPPORTED
    - The function is not supported by the driver.
- ADI_DEV_RESULT_DEVICE_IN_USE
    - The requested device is already in use.
- ADI_DEV_RESULT_NO_MEMORY
    - There is insufficient memory available.
- ADI_DEV_RESULT_BAD_DEVICE_NUMBER
    - The device number is invalid.
- ADI_DEV_RESULT_DIRECTION_NOT_SUPPORTED
    - The device cannot be opened in the direction specified.
- ADI_DEV_RESULT_BAD_DEVICE_HANDLE
    - The handle to the device driver is invalid.
- ADI_DEV_RESULT_BAD_MANAGER_HANDLE
    - The handle to the Device Manager is invalid.
- ADI_DEV_RESULT_BAD_PDD_HANDLE
    - The handle to the physical driver is invalid.
- ADI_DEV_RESULT_INVALID_SEQUENCE
    - The action requested is not within a valid sequence.
- ADI_DEV_RESULT_ATTEMPTED_READ_ON_OUTBOUND_DEVICE
    - The client attempted to provide an inbound buffer for a device opened for outbound traffic only.
- ADI_DEV_RESULT_ATTEMPTED_WRITE_ON_INBOUND_DEVICE

  - The client attempted to provide an outbound buffer for a device opened for inbound traffic only.
- ADI_DEV_RESULT_DATAFLOW_UNDEFINED
  - The dataflow method has not yet been declared.
- ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE
  - The dataflow method is incompatible with the action requested.
- ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE
  - The device does not support the buffer type provided.
- ADI_DEV_RESULT_CANT_HOOK_INTERRUPT
  - The Interrupt Manager failed to hook an interrupt handler.
- ADI_DEV_RESULT_CANT_UNHOOK_INTERRUPT
  - The Interrupt Manager failed to unhook an interrupt handler.
- ADI_DEV_RESULT_NON_TERMINATED_LIST
  - The chain of buffers provided is not NULL terminated.
- ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED
  - No callback function was supplied when it was required.
- ADI_DEV_RESULT_REQUIRES_UNIDIRECTIONAL_DEVICE
  - Requires the device be opened for either inbound or outbound traffic only.
- ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE
  - Requires the device be opened for bidirectional traffic only.

5.6.2. Device Driver Specific Return Codes

The return codes listed below are supported and processed by the device driver. These event IDs are unique to this device driver.

- ADI_SPI_RESULT_ALREADY_EXCLUSIVE
  - Exclusive access not granted
- ADI_SPI_RESULT_BAD_SLAVE_NUMBER
  - Bad slave number, expected slave number 1 to 7
- ADI_SPI_RESULT_BAD_TRANSFER_INIT_MODE
  - Bad transfer init mode value
- ADI_SPI_RESULT_BAD_WORD_SIZE
  - Bad word size (expecting 8 or 16)
- ADI_SPI_RESULT_BAD_VALUE
  - Bad value passed in (expecting TRUE or FALSE)
- ADI_SPI_RESULT_DATAFLOW_ENABLED
  - Illegal because dataflow is active
- ADI_SPI_RESULT_NO_VALID_BUFFER
  - No buffer for data transfer
- ADI_SPI_RESULT_BAD_BAUD_NUMBER
  - Bad baud rate value

6. Opening and Configuring the Device Driver

This section describes the default configuration settings for the device driver and any additional configuration settings required from the client application.

6.1. Entry Point

When opening the device driver with the adi_dev_Open() function call, the client passes a parameter to the function that identifies the specific device driver that is being opened. This parameter is called the entry point. The entry point for this driver is listed below.

- ADISPIDMAEntryPoint

6.2. Default Settings

The table below describes the default configuration settings for the device driver. If the default values are inappropriate for the given system, the application should use the command IDs listed in the table to configure the device driver appropriately. Any configuration settings not listed in the table below are undefined.

Item	Default Value	Possible Values	Command ID
Transfer mode	0x00(start transfer with read of RDBR)	0x01(start transfer with write to TDBR)	ADI_SPI_CMD_SET_TRANSFER_INIT_MODE
Send Zero	0(send last word)	1(send zeros)	ADI_SPI_CMD_SEND_ZEROS
Get more data	0(discard incoming data)	1(get more data,overwrite previous data)	ADI_SPI_CMD_SET_GET_MORE_DATA
Slave select enable	0(disable)	1(enable)	ADI_SPI_CMD_SET_PSSE
Enable MISO	0(MISO disable	1(MISO enable)	ADI_SPI_CMD_SET_MISO
Size of word	0(8 bit)	1(16 bit)	ADI_SPI_CMD_SET_WORD_SIZE
LSB first	0(MSB tx/rx first)	1(LSB tx/rx first)	ADI_SPI_CMD_SET_LSB_FIRST
Clock phase	1(slave select controlled by SW)	0(slave select controlled by HW)	ADI_SPI_CMD_SET_CLOCK_PHASE
Clock polarity	0(active high CLK)	1(active low CLK)	ADI_SPI_CMD_SET_CLOCK_POLARITY
Master/slave	0(slave)	1(master)	ADI_SPI_CMD_SET_MASTER
Open drain	0(normal)	1(open drain)	ADI_SPI_CMD_SET_OPEN_DRAIN_MASTER

Table 4 - Default Settings

6.3. Additional Required Configuration Settings

In addition to the possible overrides of the default driver settings, the device driver requires the application to specify the additional configuration information listed in the table below.

Item	Possible Values	Command ID
Dataflow method	See section 5.2	ADI_DEV_CMD_SET_DATAFLOW_METHOD

Table 5 – Additional Required Settings

7. Hardware Considerations

The SPI device driver does not require any hardware configuration. By default the SPI signals function as GPIOs and slave select signals are multiplex with other peripheral signals. When the function adi_pdd_Open() is called to open this driver , the driver will set the appropriate PORTx_FER to configure the GPIO for SPI use.
To select a slave select signal via control command ADI_SPI_CMD_ENABLE_SLAVE_SELECT, the driver will set the appropriate PORTx_FER and PORT_MUX register as necessary.

8. Operation

Care should be taken in choosing which version of the SPI driver, either the DMA-driven or interrupt-driven version, based on the needs of the application. For applications that constantly read large amounts of data from an SPI connected device, the DMA-driven version of the driver is likely a better fit. For applications that read and/or write data periodically to an SPI driven device, the interrupt-driven version may be a better fit.

8.1. DMA-Driven Considerations

The DMA-driven version of the SPI driver is best suited when large amounts of data are streamed into or out of the SPI port, a simplex type operation. For example, an ADC that constantly provides data may use the SPI port to provide that data to the processor.

To effectively use the SPI driver, the application should open the driver, configure the driver for the intended mode of operation and configuration parameters, provide either inbound buffers to the adi_dev_Read() function when data will be streamed into the Blackfin or outbound buffers to the adi_dev_Write() function when data will be streamed out of the Blackfin.

8.1.1. Inbound Example

The code below illustrates how to use the DMA-driven version of the SPI driver to stream data into the Blackfin. This example uses the chained with loopback dataflow method. By using this method, buffers are provided when the driver is initialized and then used over and over again without having to be resubmitted.

```
#define BUFFER_COUNT (4)
#define DATA_COUNT   (512)

static u16                    Data[DATA_COUNT * BUFFER_COUNT];
static ADI_DEV_1D_BUFFER      Buffer[BUFFER_COUNT];
static ADI_DEV_DEVICE_HANDLE  SPIHandle;


/*********************************************************************

    Function:       Callback

    Description:    Process data that has been received

*********************************************************************/

static void Callback(void *AppHandle, u32 Event, void *pArg)
{

    ADI_DEV_1D_BUFFER   *pBuffer;

    // CASEOF (Event)
    switch (Event) {

        // CASE (Buffer processed)
        case ADI_DEV_EVENT_BUFFER_PROCESSED:
            // avoid casts
            pBuffer = pArg;

            // process the data that was received
            ProcessBuffer(pBuffer);
            break;

    // ENDCASE
    }

    // return
}


/*********************************************************************

    Function:     DriverApp

    Description:  Open, configures, starts driver

*********************************************************************/

void DriverApp(void) {

    ADI_DEV_CMD_VALUE_PAIR SPIConfig[] = {
        { ADI_DEV_CMD_SET_DATAFLOW_METHOD, (void *)ADI_DEV_MODE_CHAINED_LOOPBACK },  // chained loopback
        { ADI_SPI_CMD_SET_BAUD_REG,        (void *)0x7ff                         },  // run at some rate
        { ADI_SPI_CMD_SET_WORD_SIZE,       (void *)16                            },  // word size 16 bits
        { ADI_SPI_CMD_SET_MASTER,          (void *)TRUE                          },  // master mode
        { ADI_SPI_CMD_ENABLE_SLAVE_SELECT, (void *)4                             },  // device is on SS 4
        { ADI_SPI_CMD_SELECT_SLAVE,        (void *)4                             },  // enable the device
        { ADI_DEV_CMD_END,                 NULL                                  }
    };

    u32 Result;
    int i;

    // open DMA-driven SPI driver
    Result = adi_dev_Open(adi_dev_ManagerHandle, &ADISPIDMAEntryPoint, 0, 0, &SPIHandle,
                          ADI_DEV_DIRECTION_INBOUND, NULL, NULL, Callback);

    // configure SPI
    Result = adi_dev_Control(SPIHandle, ADI_DEV_CMD_TABLE, SPIConfig);

    // prepare buffers
    for (i = 0; i < BUFFER_COUNT; i++) {
        Buffer[i].Data              = &Data[i * DATA_COUNT];
        Buffer[i].ElementCount      = DATA_COUNT;
        Buffer[i].ElementWidth      = 2;
        Buffer[i].CallbackParameter = &Buffer[i];
        Buffer[i].pNext             = &Buffer[i+1];
    }
    Buffer[BUFFER_COUNT-1].pNext = NULL;

    // give the driver the buffers
    Result = adi_dev_Read(SPIHandle, ADI_DEV_1D, (ADI_DEV_BUFFER *)Buffer);

    // enable dataflow
    Result = adi_dev_Control(SPIHandle, ADI_DEV_CMD_SET_DATAFLOW, (void *)TRUE);

    // spin
    while (1) ;
}
```

8.1.2. Outbound Example

The code below illustrates how to use the DMA-driven version of the SPI driver to stream data out from the Blackfin. This example is virtually identical to the inbound one except adi_dev_Write() is called instead of adi_dev_Read().

Again, by using the chained with loopback method, buffers are provided when the driver is initialized then used over and over without needing to be resubmitted.

```
#define BUFFER_COUNT (4)
#define DATA_COUNT   (512)

static u16                    Data[DATA_COUNT * BUFFER_COUNT];
static ADI_DEV_1D_BUFFER      Buffer[BUFFER_COUNT];
static ADI_DEV_DEVICE_HANDLE  SPIHandle;


/*********************************************************************

    Function:       Callback

    Description:    Fill buffers up with more data to send out

*********************************************************************/

static void Callback(void *AppHandle, u32 Event, void *pArg)
{

    ADI_DEV_1D_BUFFER   *pBuffer;

    // CASEOF (Event)
    switch (Event) {

        // CASE (Buffer processed)
        case ADI_DEV_EVENT_BUFFER_PROCESSED:

            // avoid casts
            pBuffer = pArg;

            // fill the buffer up with new data
            FillBuffer(pBuffer);
            break;

    // ENDCASE
    }

    // return
}


/*********************************************************************

    Function:     DriverApp

    Description:  Open, configures, starts driver

*********************************************************************/

void DriverApp(void) {

    ADI_DEV_CMD_VALUE_PAIR SPIConfig[] = {
        { ADI_DEV_CMD_SET_DATAFLOW_METHOD, (void *)ADI_DEV_MODE_CHAINED_LOOPBACK },  // chained loopback
        { ADI_SPI_CMD_SET_BAUD_REG,        (void *)0x7ff                         },  // run at some rate
        { ADI_SPI_CMD_SET_WORD_SIZE,       (void *)16                            },  // word size 16 bits
        { ADI_SPI_CMD_SET_MASTER,          (void *)TRUE                          },  // master mode
        { ADI_SPI_CMD_ENABLE_SLAVE_SELECT, (void *)4                             },  // device is on SS 4
        { ADI_SPI_CMD_SELECT_SLAVE,        (void *)4                             },  // enable the device
        { ADI_DEV_CMD_END,                 NULL                                  }
    };

    u32 Result;
    int i;

    // open DMA-driven SPI driver
    Result = adi_dev_Open(adi_dev_ManagerHandle, &ADISPIDMAEntryPoint, 0, 0, &SPIHandle,
                      ADI_DEV_DIRECTION_OUTBOUND, NULL, NULL, Callback);

    // configure SPI
    Result = adi_dev_Control(SPIHandle, ADI_DEV_CMD_TABLE, SPIConfig);

    // prepare buffers
    for (i = 0; i < BUFFER_COUNT; i++) {
        Buffer[i].Data              = &Data[i * DATA_COUNT];
        Buffer[i].ElementCount      = DATA_COUNT;
        Buffer[i].ElementWidth      = 2;
        Buffer[i].CallbackParameter = &Buffer[i];
        Buffer[i].pNext             = &Buffer[i+1];
    }
    Buffer[BUFFER_COUNT-1].pNext = NULL;

    // give the driver the buffers
    Result = adi_dev_Write(SPIHandle, ADI_DEV_1D, (ADI_DEV_BUFFER *)Buffer);

    // enable dataflow
    Result = adi_dev_Control(SPIHandle, ADI_DEV_CMD_SET_DATAFLOW, (void *)TRUE);

    // spin
    while (1) ;
}
```