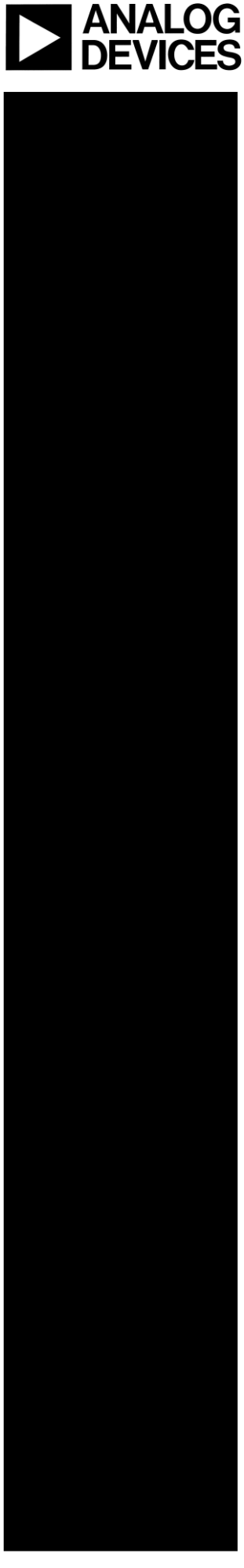

ADI_SPORT
DEVICE DRIVER

DATE: DECEMBER 20, 2005

Table of Contents

Table of Figures

Document Revision History

Date	Description of Changes
2005-12-20	Initial revision
2006-01-04	Minor edits – added in entry point definition
2006-05-18	Corrected file locations
2007-01-16	Added mandatory requirement info to ‘Set Tx/Rx Frame sync frequency’ commands
05 Jan 2010	Updated with command to set SPORT Pin Mux
2010-10-14	Updated with command to add hysteresis to the SPORT input pins

Table 1 - Revision History

1. Overview

This document describes the usage of the SPORT device driver. As written, the driver has no hardware dependencies and has been tested on the ADSP-BF533, ADSP-BF537 and ADSP-BF561 EZ-Kit Lite development boards.

The SPORT device driver supports all major modes of the SPORT including stereo mode, similar to I2S, and multi-channel operation. The SPORT driver uses DMA for data movement, leveraging the DMA Manager system service, allowing maximum throughput via the SPORT device.

Unlike most other device drivers, under certain conditions, the SPORT device driver allows a single SPORT peripheral to be opened twice. If a SPORT has been opened for bidirectional dataflow, then both inbound and outbound channels are used so the driver will not allow any other clients to open that same SPORT device. If however, a SPORT has been opened for either inbound only or outbound only dataflow, then the driver allows the SPORT to be opened again, as a separate device, in the opposite dataflow direction. For example, if SPORT0 was opened for inbound data, SPORT0 could be opened once again for outbound data; effectively presenting two different devices even though there is only a single physical device.

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/sport/adi_sport.h> This file contains all commands, event and return codes specific to the SPORT device driver.

2.2. Source Files

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

- <Blackfin/lib/src/drivers/sport/adi_sport.c> This file contains all source code for the SPORT device driver. All source code is written in 'C'. There are no assembly level functions in the driver.

3. Lower Level Drivers

The SPORT device driver does not use any lower level device drivers.

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

Unless overridden with the appropriate commands, the SPORT device driver uses the default Interrupt Vector Group (IVG) mappings and DMA peripheral mappings. If changes to any of these settings from the default configuration are desired, those changes should be made prior to opening the SPORT driver. Changes to the IVG mappings can be accomplished by appropriate calls into the Interrupt Manager service while changes to the DMA peripheral mappings can be affected by appropriate calls into the DMA Manager service.

Operation of the SPORT device driver may involve the use of up to 4 interrupts; transmit, receive, SPORT error and DMA error. Because the SPORT device driver uses the DMA Manager service for data movement, the enabling/disabling of interrupts and hooking/unhooking of interrupt handlers for the transmit, receive and DMA error, are performed by the DMA Manager. The SPORT driver itself controls enabling/disabling and hooking/unhooking of handlers for the SPORT error interrupt. Control of these interrupts is as follows:

- Transmit (Tx) Interrupt – When the SPORT driver is opened for outbound or bidirectional traffic, the DMA Manager enables and hooks the interrupt for the DMA channel to which the SPORT Tx peripheral is mapped. The DMA Manager disables and unhooks the DMA channel interrupt when the SPORT driver is closed.
- Receive (Rx) Interrupt – When the SPORT driver is opened for inbound or bidirectional traffic, the DMA Manager enables and hooks the interrupt for the DMA channel to which the SPORT Rx peripheral is mapped. The DMA Manager disables and unhooks the DMA channel interrupt when the SPORT driver is closed.
- DMA Error Interrupt – When the SPORT driver is opened for any direction, the DMA Manager enables and hooks the DMA error interrupt. The interrupt is disabled and unhooked when the SPORT driver is closed and no other clients of the DMA Manager are using that same DMA error interrupt.
- SPORT Error Interrupt – This interrupt is enabled/disabled and hooked/unhooked when the client passes the ADI_DEV_CMD_SET_ERROR_REPORTING command to the driver. By default, SPORT error interrupts are not generated so if SPORT error interrupts are desired they must be explicitly turned on with the command. When the adi_dev_Control() function receives the command with an accompanying argument of TRUE, the SPORT error interrupt is enabled and the interrupt handler is hooked into the IVG chain. When the adi_dev_Control() function receives the command with an accompanying argument of FALSE, the SPORT error interrupt is disabled and the interrupt handler is unhooked from the IVG chain. If enabled and hooked, the SPORT error interrupt is disabled and the interrupt handler is automatically unhooked when the driver is closed (adi_dev_Close()).

4.2. DMA

The SPORT device driver leverages the services of the DMA Manager for transfer of all data through the SPORT. When opened for inbound data traffic, only the SPORT Rx DMA channel is used. When opened for outbound DMA data traffic, only the SPORT Tx DMA channel is used. When opened for bidirectional dataflow, both the SPORT Rx and SPORT Tx DMA channels are used.

4.3. Timers

The SPORT device driver does not use any timer resources.

4.4. Real-Time Clock

The SPORT device driver does not use any RTC resources.

4.5. Programmable Flags

The SPORT device driver does not explicitly use any programmable flag pins. However, on most processors the SPORT pins are muxed with other peripherals including programmable flag pins. The user should use caution to insure that pins used by the SPORT do not interfere with any general purpose I/O use and vice-versa.

4.6. Pins

Each SPORT provides 8 pins. These pins are:

- TSCLKx Transmit serial clock
- TFSx Transmit frame sync
- DTxPRI Primary transmit data
- DTxSEC Secondary transmit data
- RSCLKx Receive serial clock
- RFSx Receive frame sync
- DRxPRI Primary receive data
- DRxSEC Secondary receive data

On processors where pin muxing is used, the SPORT device driver uses the Port Control service to automatically configure pins for use by the SPORT as described below:

- Bidirectional data flow – all SPORT pins configured for SPORT usage.
- Inbound only data flow – RSCLKx, RFSx, DRxPRI and DRxSEC are configured for SPORT usage.
- Outbound only data flow – TSCLKx, TFSx, DTxPRI and DTxSEC are configured for SPORT usage.

Pins are automatically configured when dataflow is enabled on the SPORT by the ADI_DEV_CMD_SET_DATAFLOW command.

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.
ADI_DEV_ DIRECTION_BIDIRECTIONAL	Supports both the reception of data and transmission of data 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_CIRCULAR	Supports the circular buffer method
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_CIRCULAR_BUFFER
  - Circular buffer
  - pAdditionalInfo – ignored
- ADI_DEV_1D_BUFFER
  - Linear one-dimensional buffer
  - pAdditionalInfo – ignored
- ADI_DEV_2D_BUFFER
  - Two-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.

5.4.3.1. SPORT Transmit Control Commands

- ADI_SPORT_CMD_CLEAR_TX_ERRORS
  - Clears all transmit error conditions
  - Value – NULL
- ADI_SPORT_CMD_SET_TX_CLOCK_FREQ
  - Sets the transmit data clock frequency
  - Value – u32 (frequency in Hz)
- ADI_SPORT_CMD_SET_TX_FRAME_SYNC_FREQ
  - Sets the transmit frame sync clock frequency. Client must set Tx Clock frequency before setting Tx Frame Sync frequency.
  - Value – u32 (frequency in Hz)
- ADI_SPORT_CMD_SET_TX_CLOCK_SOURCE
  - Sets the transmit clock source
  - Value – u16 (0 – external, 1 – internal)
- ADI_SPORT_CMD_SET_TX_DATA_FORMAT
  - Sets the transmit data format
  - Value – u16 (0 – normal, 1 – reserved, 2 – ulaw, 3 – Alaw)
- ADI_SPORT_CMD_SET_TX_BIT_ORDER
  - Sets the transmit bit order
  - Value – u16 (0 – MSB first, 1 – LSB first)
- ADI_SPORT_CMD_SET_TX_FS_SOURCE
  - Sets the transmit frame sync source
  - Value – u16 (0 – external, 1 – internal)
- ADI_SPORT_CMD_SET_TX_FS_REQUIREMENT
  - Sets the frame sync requirement
  - Value – u16 (0 – no transmit frame sync with each word, 1 – transmit frame sync for each word)
- ADI_SPORT_CMD_SET_TX_FS_DATA_GEN
  - Sets the transmit data based frame sync generation
  - Value – u16 (0 – data dependent frame sync generation, 1 – data independent frame sync generation)
- ADI_SPORT_CMD_SET_TX_FS_POLARITY
  - Sets the transmit frame sync polarity
  - Value – u16 (0 – active high frame sync, 1 – active low frame sync)
- ADI_SPORT_CMD_SET_TX_FS_TIMING
  - Sets the transmit frame sync timing
  - Value – u16 (0 – early frame sync, 1 – late frame sync)
- ADI_SPORT_CMD_SET_TX_EDGE_SELECT
  - Sets the transmit edge selection

    - Value – u16 (0 – data and internal frame sync on rising edge of TSCLK and external frame sync on falling edge, 1 – data and internal frame sync on falling edge of TSCLK and external frame sync on rising edge)
- ADI_SPORT_CMD_SET_TX_WORD_LENGTH
    - Sets the transmit word length
    - Value – u16 (1 less than the serial word length (in bits), values of 0, 1 and greater than 31 are illegal)
- ADI_SPORT_CMD_SET_TX_SECONDARY_ENABLE
    - Enables/disables the secondary transmit data channel
    - Value – (TRUE – enabled, FALSE – disabled)
- ADI_SPORT_CMD_SET_TX_STEREO_FS_ENABLE
    - Enables/disabled the transmit stereo frame sync
    - Value – (TRUE – frame sync is left/right clock, FALSE – normal mode)
- ADI_SPORT_CMD_SET_TX_LEFT_RIGHT_ORDER
    - Sets the left/right channel order
    - Value – u16 (0 – left channel first, 1 – right channel first)

5.4.3.2. SPORT Receive Control Commands

- ADI_SPORT_CMD_CLEAR_RX_ERRORS
    - Clears all receive error conditions
    - Value – NULL
- ADI_SPORT_CMD_SET_RX_CLOCK_FREQ
    - Sets the receive data clock frequency
    - Value – u32 (frequency in Hz)
- ADI_SPORT_CMD_SET_RX_FRAME_SYNC_FREQ
    - Sets the receive frame sync clock frequency. Client must set Rx Clock frequency before setting Rx Frame Sync frequency.
    - Value – u32 (frequency in Hz)
- ADI_SPORT_CMD_SET_RX_CLOCK_SOURCE
    - Sets the receive clock source
    - Value – u16 (0 – external, 1 – internal)
- ADI_SPORT_CMD_SET_RX_DATA_FORMAT
    - Sets the receive data format
    - Value – u16 (0 – zero fill, 1 – sign extend, 2 – ulaw, 3 – Alaw)
- ADI_SPORT_CMD_SET_RX_BIT_ORDER
    - Sets the receive bit order
    - Value – u16 (0 – MSB first, 1 – LSB first)
- ADI_SPORT_CMD_SET_RX_FS_SOURCE
    - Sets the receive frame sync source
    - Value – u16 (0 – external, 1 – internal)
- ADI_SPORT_CMD_SET_RX_FS_REQUIREMENT
    - Sets the frame sync requirement
    - Value – u16 (0 – no receive frame sync with each word, 1 – receive frame sync for each word)
- ADI_SPORT_CMD_SET_RX_FS_POLARITY
    - Sets the receive frame sync polarity
    - Value – u16 (0 – active high frame sync, 1 – active low frame sync)
- ADI_SPORT_CMD_SET_RX_FS_TIMING
    - Sets the receive frame sync timing
    - Value – u16 (0 – early frame sync, 1 – late frame sync)
- ADI_SPORT_CMD_SET_RX_EDGE_SELECT
    - Sets the receive edge selection
    - Value – u16 (0 – data and internal frame sync on rising edge of RSCLK and external frame sync on falling edge, 1 – data and internal frame sync on falling edge of RSCLK and external frame sync on rising edge)
- ADI_SPORT_CMD_SET_RX_WORD_LENGTH
    - Sets the receive word length
    - Value – u16 (1 less than the serial word length (in bits), values of 0, 1 and greater than 31 are illegal)

- ADI_SPORT_CMD_SET_RX_SECONDARY_ENABLE
  - Enables/disables the secondary receive data channel
  - Value – (TRUE – enabled, FALSE – disabled)
- ADI_SPORT_CMD_SET_RX_STEREO_FS_ENABLE
  - Enables/disabled the receive stereo frame sync
  - Value – (TRUE – frame sync is left/right clock, FALSE – normal mode)
- ADI_SPORT_CMD_SET_RX_LEFT_RIGHT_ORDER
  - Sets the left/right channel order
  - Value – u16 (0 – left channel first, 1 – right channel first)

5.4.3.3. SPORT Multi-Channel Control Commands

- ADI_SPORT_CMD_SET_MC_WINDOW_OFFSET
  - Sets the window offset
  - Value – u16 (0 to 1023)
- ADI_SPORT_CMD_SET_MC_WINDOW_SIZE
  - Sets the window size
  - Value – u16 ((window size /8) – 1), i.e. for a window size of 512 the value should be 63.
- ADI_SPORT_CMD_SET_MC_CLOCK_RECOVERY_MODE
  - Sets the 2x clock recovery mode
  - Value – u16 (0 or 1 – bypass, 2 – recover 2 MHz clock from 4 MHz clock, 3 – recover 8 MHz clock from 16 MHz clock)
- ADI_SPORT_CMD_SET_MC_MODE
  - Sets the multi-channel mode
  - Value – (TRUE – enabled, FALSE – disabled)
- ADI_SPORT_CMD_SET_MC_FS_TO_DATA
  - Sets the frame sync to data relationship
  - Value – u16 (0 – Normal, 1 – reversed (H.100 mode))
- ADI_SPORT_CMD_SET_MC_FRAME_DELAY
  - Sets the multi-channel frame delay
  - Value – u16 (0 to 15 cycles)
- ADI_SPORT_CMD_SET_MC_TX_PACKING
  - Sets the transmit packing
  - Value – (TRUE – enabled, FALSE – disabled)
- ADI_SPORT_CMD_SET_MC_TX_CHANNEL_ENABLE
  - Enables a transmit channel
  - Value – u16 (channel number between 0 and 127)
- ADI_SPORT_CMD_SET_MC_TX_CHANNEL_DISABLE
  - Disables a transmit channel
  - Value – u16 (channel number between 0 and 127)
- ADI_SPORT_CMD_SET_MC_RX_PACKING
  - Sets the receive packing
  - Value – (TRUE – enabled, FALSE – disabled)
- ADI_SPORT_CMD_SET_MC_RX_CHANNEL_ENABLE
  - Enables a receive channel
  - Value – u16 (channel number between 0 and 127)
- ADI_SPORT_CMD_SET_MC_RX_CHANNEL_DISABLE
  - Disables a receive channel
  - Value – u16 (channel number between 0 and 127)

5.4.3.4. SPORT Register commands

5.4.3.4.1. Transmit Registers

- ADI_SPORT_CMD_SET_TCR1

    - Sets the TCR1 register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_TCR2
    - Sets the TCR2 register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_TCLKDIV
    - Sets the TCLKDIV register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_TFSDIV
    - Sets the TFSDIV register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_TX16
    - Sets the TX16 register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_TX32
    - Sets the TX32 register
    - Value – u32 (register value)

5.4.3.4.2. Receive Registers

- ADI_SPORT_CMD_SET_RCR1
    - Sets the RCR1 register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_RCR2
    - Sets the RCR2 register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_RCLKDIV
    - Sets the RCLKDIV register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_RFSDIV
    - Sets the RFSDIV register
    - Value – u16 (register value)
- ADI_SPORT_CMD_GET_RX16
    - Gets the RX16 register
    - Value – u16 * (location where contents of RX16 will be stored)
- ADI_SPORT_CMD_GET_RX32
    - Gets the RX32 register
    - Value – u32 * (location where contents of RX32 will be stored)

5.4.3.4.3. Status Registers

- ADI_SPORT_CMD_GET_STAT
    - Gets the STAT register
    - Value – u16 * (location where contents of STAT will be stored)

5.4.3.4.4. Multi-Channel Registers

- ADI_SPORT_CMD_SET_MCMC1
    - Sets the MCMC1 register
    - Value – u16 (register value)
- ADI_SPORT_CMD_SET_MCMC2
    - Sets the MCMC2 register
    - Value – u16 (register value)
- ADI_SPORT_CMD_GET_CHNL
    - Gets the CHNL register
    - Value – u16 * (location where contents of CHNL will be stored)

- ADI_SPORT_CMD_SET_MTCS0
  - Sets the MTCS0 register
  - Value – u16 (register value)
- ADI_SPORT_CMD_SET_MTCS1
  - Sets the MTCS1 register
  - Value – u16 (register value)
- ADI_SPORT_CMD_SET_MTCS2
  - Sets the MTCS2 register
  - Value – u16 (register value)
- ADI_SPORT_CMD_SET_MTCS3
  - Sets the MTCS3 register
  - Value – u16 (register value)
- ADI_SPORT_CMD_SET_MRCS0
  - Sets the MRCS0 register
  - Value – u16 (register value)
- ADI_SPORT_CMD_SET_MRCS1
  - Sets the MRCS1 register
  - Value – u16 (register value)
- ADI_SPORT_CMD_SET_MRCS2
  - Sets the MRCS2 register
  - Value – u16 (register value)
- ADI_SPORT_CMD_SET_MRCS3
  - Sets the MRCS3 register
  - Value – u16 (register value)

- ADI_SPORT_CMD_SET_PIN_MUX_MODE
  - Sets processor specific pin mux mode
  - Value = Enumeration of type ADI_SPORT_PIN_MUX_MODE

Processor Family	ADI_SPORT_PIN_MUX_MODE Enumeration value	Comments
ADSP - BF50x (Moy)	ADI_SPORT_PIN_MUX_MODE_0	SPORT Pin Mux configuration mode 0 (default) For SPORT 1 - PG4 as Secondary Rx channel (DR1SEC)
	ADI_SPORT_PIN_MUX_MODE_1	SPORT Pin Mux configuration mode 1 For SPORT 1 - PG8 as Secondary Rx channel (DR1SEC)
Other Processors	Command Not supported	

- ADI_SPORT_CMD_SET_HYSTERESIS_ENABLE (ADSP-BF53x and ADSP-BF561 only)
  - Enables/disables hysteresis on the SPORT input pins.
  - Value – TRUE/FALSE

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_SUB_BUFFER_PROCESSED
  - Notifies callback function that a sub-buffer within a circular buffer has been processed by the device driver.
  - Value – 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_SPORT_EVENT_ERROR_INTERRUPT
  - Notifies the callback function that the SPORT generated an error interrupt.
  - Value – NULL

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_SPORT_RESULT_BAD_ACCESS_WIDTH
    - Access to one of the TX16, TX32, RX16 or RX32, registers was attempted when the SPORT word length field (SLEN in the appropriate control register), did not match the width of the register being accessed.

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.

- ADISPORTEntryPoint

6.2. Default Settings

When the SPORT is opened via the adi_dev_Open() function call, the device driver resets the SPORT control and status registers to their default, power-up value, of all zeros as indicated in the table below

Data Direction	Registers Reset to 0	Error Conditions Cleared
ADI_DEV_DIRECTION_INBOUND	RCR1, RCR2, RCLKDIV, RFSDIV, MRCS0, MRCS1, MRCS2, MRCS3, MCMC1, MCMC2	All receive errors
ADI_DEV_DIRECTION_OUTBOUND	TCR1, TCR2, TCLKDIV, TFSDIV, MTCS0, MTCS1, MTCS2, MTCS3, MCMC1, MCMC2	All transmit errors
ADI_DEV_DIRECTION_BIDIRECTIONAL	RCR1, RCR2, RCLKDIV, RFSDIV, MRCS0, MRCS1, MRCS2, MRCS3, TCR1, TCR2, TCLKDIV, TFSDIV, MTCS0, MTCS1, MTCS2, MTCS3, MCMC1, MCMC2	All receive and transmit errors

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

There are no special hardware considerations for the SPORT device driver.