ANALOG DEVICES

ADI_AD1836A_II
DEVICE DRIVER

DATE: SEPTEMBER 5, 2006.

Table of Contents

1. Overview 6

2. Files 7

2.1. Include Files 7

2.2. Source Files 7

3. Lower Level Drivers 8

3.1. SPI Device Driver 8

3.2. SPORT Device Driver 8

4. Resources Required 9

4.1. Interrupts 9

4.2. DMA 9

4.3. Timers 9

4.4. Real-Time Clock 9

4.5. Programmable Flags 9

4.6. Pins 9

5. Supported Features of the Device Driver 10

5.1. Directionality 10

5.2. Dataflow Methods 10

5.3. Buffer Types 10

5.4. Command IDs 10

5.4.1. Device Manager Commands 11

5.4.2. Common Commands 11

5.4.3. Device Driver Specific Commands 13

5.5. Callback Events 13

5.5.1. Common Events 13

5.5.2. Device Driver Specific Events 14

5.6. Return Codes 14

5.6.1. Common Return Codes 14

5.6.2. Device Driver Specific Return Codes 15

5.7. Auto-SPORT Configuration 16

6. Configuring the Device Driver 17

6.1. Entry Point 17

6.2. Default Settings 17

6.3. Additional Required Configuration Settings 17

7. Hardware Considerations 18

7.1. AD1836A registers 18

7.2. AD1836A register fields 18

8. Appendix 20

8.1. Using AD1836Aii Device Driver in Applications 20

8.1.1. Device Manager Data memory allocation 20

8.1.2. DMA Manager Data memory allocation 20

8.1.3. Typical usage of AD1836A device driver 20

8.1.4. Re-use/share the SPORT device reserved by AD1836A 21

8.1.5. Resetting AD1836A device 21

8.2. Accessing AD1836A registers 23

8.2.1. Read AD1836A internal registers 23

8.2.2. Configure AD1836A internal registers 25

List of Tables

Table 1 – Revision History 5
Table 2 – Supported Dataflow Directions 10
Table 3 – Supported Dataflow Methods 10
Table 4 – Default SPORT configuration in auto-SPORT config mode 16
Table 5 – Default Settings 17
Table 6 – Additional Required Settings 17
Table 7 – AD1836A device registers 18
Table 8 – AD1836A Register Fields 19

Document Revision History

Date	Description of Changes
2006/01/24	Initial release
2006/05/16	Updated to new device access interface Added register access examples
2006/09/05	Added details on ‘auto-SPORT config’ mode

Table 1 – Revision History

1. Overview

The driver allows user to control AD1836A Audio Codec. The codec's sub-address registers are accessed via SPI port and the audio dataflow is through SPORT. The application program can access internal registers of the codec using device access commands and specific return codes are sent in result success or failure. This driver can sense any change in AD1836A device settings and automatically update the corresponding SPORT device configuration in relevance to present AD1836A operating mode.

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 definitions, function prototypes etc. specific to SPORT device
- <drivers/deviceaccess/adi_device_access.h> This file contains all definitions, function prototypes etc. for TWI/SPI device access service
- <drivers/codec/adi_ad1836a_ii.h > This file contains all definitions, function prototypes etc. specific to AD1836A device

2.2. Source Files

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

- adi_ad1836a_ii.c

3. Lower Level Drivers

AD1836A driver is layered on SPI and SPORT drivers.

3.1. SPI Device Driver

AD1836A can be operated in various modes by configuring its internal registers and it is done via SPI serial port. By default, AD1836A device driver sets 0x04 as its default SPI device address.

3.2. SPORT Device Driver

Serial Port (SPORT) is used for audio data transfer to and from the codec. By default, AD1836A device driver uses SPORT device 0 for its audio dataflow.

Application can directly communicate with the SPORT device allocated for AD1836A audio dataflow by calling adi_dev_Control( ) function with AD1836A driver Handle, SPORT driver specific command the and value specific to the command. Before issuing any such SPORT specific commands, application must open the SPORT device allocated to AD1836A. Refer to section 8.1.3 for more details.

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()).

Wherever possible, this device driver uses the System Services to perform the necessary low-level hardware access and control.

The AD1836A device driver is build upon interrupt driven SPI driver and DMA operated SPORT driver.

4.1. Interrupts

No specific interrupts or interrupt handlers are used by this driver.

4.2. DMA

The driver doesn't support DMA directly, but uses a DMA driven SPORT for its audio dataflow. AD1836A supports bi-directional dataflow and enough memory should be allocated for the DMA channels depending on the codec's audio dataflow method. If the client intends to use a DMA driven SPI, enough memory should be allocated for that DMA channel as well.

4.3. Timers

Timer service is not used by this driver.

4.4. Real-Time Clock

RTC service is not used by this driver

4.5. Programmable Flags

No Programmable Flags are used by this driver

4.6. Pins

Connect corresponding SPORT device port pins to serial data I/O port pins of AD1836A.
Connect corresponding SPI device port pins of Blackfin processor to Control port pins of AD1836A.

Refer to corresponding device reference manuals for further information.

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_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_REGISTER_READ
  - Reads a single device register
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics)
- ADI_DEV_CMD_REGISTER_FIELD_READ
  - Reads a specific field location in a single device register
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)
- ADI_DEV_CMD_REGISTER_TABLE_READ
  - Reads a table of selective device registers
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics)
- ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ
  - Reads a table of selective device register fields
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)
- ADI_DEV_CMD_REGISTER_BLOCK_READ
  - Reads a block of consecutive device registers
  - Value – ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics)
- ADI_DEV_CMD_REGISTER_WRITE
  - Writes to a single device register
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics)
- ADI_DEV_CMD_REGISTER_FIELD_WRITE
  - Writes to a specific field location in a single device register
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)
- ADI_DEV_CMD_REGISTER_TABLE_WRITE
  - Writes to a table of selective device registers
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics)

- ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE
  - Writes to a table of selective device register fields
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)
- ADI_DEV_CMD_REGISTER_BLOCK_WRITE
  - Writes to a block of consecutive device registers
  - Value – ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics)

This driver also supports following commands and all SPORT specific commands, provided the SPORT device allocated for AD1836A audio dataflow is opened before issuing any of these commands. Refer to SPORT driver documentation for details on SPORT specific commands

- 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_AD1836A_CMD_SET_SPI_CS
  - Sets SPI Chip-select for AD1836A
  - Value – u8
- ADI_AD1836A_CMD_GET_SPI_CS
  - Gets present SPI Chip-select used for AD1836A
  - Value – u8 *
- ADI_AD1836A_CMD_SET_SPORT_DEVICE_NUMBER
  - Sets SPORT device number to be used for AD1836A audio dataflow.
  - Value – u8
- ADI_AD1836A_CMD_SET_SPORT_STATUS
  - Sets status of SPORT device to be used for AD1836A audio dataflow (Opens/Closes SPORT device). Application MUST issue this command before passing any SPORT driver specific commands.
  - Value – ADI_AD1836A_SET_SPORT_STATUS
- ADI_AD1836A_CMD_ENABLE_AUTO_SPORT_CONFIG
  - Enable(TRUE) / Disable(FALSE) auto-SPORT configuration mode
  - Value - TRUE/FALSE (TRUE by default)
- ADI_AD1836A_CMD_SET_SPORT_OPERATION_MODE
  - Command no longer used / supported by this driver. Left for backward compatibility

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 client should prepare its callback function to process each event described in these two 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.

This driver doesn't have any unique events.

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. The client should prepare to process each of the return codes described in these sections.

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.

Return codes specific to TWI/SPI Device access service

- ADI_DEV_RESULT_CMD_NOT_SUPPORTED
  - Command not supported by the Device Access Service
- ADI_DEV_RESULT_INVALID_REG_ADDRESS
  - The client attempting to access an invalid register address
- ADI_DEV_RESULT_INVALID_REG_FIELD
  - The client attempting to access an invalid register field location
- ADI_DEV_RESULT_INVALID_REG_FIELD_DATA
  - The client attempting to write an invalid data to selected register field location
- ADI_DEV_RESULT_ATTEMPT_TO_WRITE_READONLY_REG
  - The client attempting to write to a read-only location
- ADI_DEV_RESULT_ATTEMPT_TO_ACCESS_RESERVE_AREA
  - The client attempting to access a reserved location
- ADI_DEV_RESULT_ACCESS_TYPE_NOT_SUPPORTED
  - Device Access Service does not support the access type provided by the driver

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_AD1836A_RESULT_CMD_NOT_SUPPORTED
  - Command supplied by the client is not supported by AD1836A device driver

5.7. Auto-SPORT Configuration

This driver can sense any change in AD1836A device register settings (usually carried out by the application) and automatically update the corresponding SPORT device settings to support present AD1836A mode (this feature is referred as 'Auto-SPORT config' mode).

'Auto SPORT config' mode is NOT supported when AD1836A is operated in Right Justified, Packed 128 or Packed 256 mode. Any application intend to operate AD1836A in any of these modes can configure the SPORT device by issuing SPORT driver specific commands with AD1836A driver handle.

Auto-SPORT config mode can be enabled / disabled by issuing command ADI_AD1836A_CMD_ENABLE_AUTO_SPORT_CONFIG with value as TRUE/FALSE.

This driver enables Auto-SPORT config mode by default.

AD1836A mode	SPORT Configuration	SPORT Primary channel	SPORT Secondary channel
I2S mode (16/20/24 bit word length)	• TFSR,RFSR enabled • TCKFE, RCKFE enabled • TSFSE, RSFSE enabled • SLEN set similar to AD1836A operating word length	Both Tx and Rx Primary channels are enabled by default	Both Tx and Rx Secondary channels are enabled by default
Left-Justified mode (16/20/24 bit word length)	• LATFS, LARFS enabled • LTFS, LRFS enabled • TFSR, RFSR enabled • TSFSE, RSFSE enabled • SLEN set similar to AD1836A operating word length	Both Tx and Rx Primary channels are enabled by default	Both Tx and Rx Secondary channels are enabled by default
DSP mode (16/20/24 bit word length)	• DITFS enabled • TFSR,RFSR enabled • SLEN set similar to AD1836A operating word length	Both Tx and Rx Primary channels are enabled by default	Both Tx and Rx Secondary channels are enabled by default
Packed mode AUX (TDM) (16/20/24 bit word length)	• TFSR,RFSR enabled • Multichannel enabled • DMA Tx/Rx packing enabled • 8 Transmit channels enabled • 8 Receive channels enabled • 8 active channels starts from window 0 • SLEN set to 32 regardless of AD1836A operating word length	Both Tx and Rx Primary channels are enabled by default	Both Tx and Rx Secondary channels are disabled by default

Table 4 – Default SPORT configuration in auto-SPORT config mode

****Note****
Whenever the application switches from Packed mode Aux to other (serial) mode, application should first change AD1836A ADC serial mode before modifying the DAC serial mode. The reason is, AD1836A ADC in Packed mode Aux bypasses DAC serial mode settings and any modifications to DAC serial mode will be ignored by the driver as well as by AD1836A device.

6. 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.

- ADIAD1836AEntryPoint

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.

Item	Default Value	Possible Values	Command ID
SPORT Device	0	Depends on number of SPORT devices available on the Blackfin processor	ADI_AD1836A_CMD_SET_SPORT_DEVICE_NUMBER

Table 5 – 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
SPI Chipselect	between 1 and 7	ADI_AD1836A_CMD_SET_SPI_CS
SPORT status	ADI_AD1836A_SPORT_OPEN, ADI_AD1836A_SPORT_CLOSE	ADI_AD1836A_CMD_SET_SPORT_STATUS

Table 6 – Additional Required Settings

7. Hardware Considerations

The client should set the SPI chip-select value (configure SPI_FLG) corresponding to AD1836A before configuring the codec to specific mode. Command id 'ADI_AD1836A_CMD_SET_SPI_CS' can be used to set AD1836A's chip-select.

7.1. AD1836A registers

The following table is a list of registers that can be accessed on the AD1836A. Refer to the AD1836A hardware reference manual for register description and chip functionality.

Register	Address	Default	Description
AD1836A_DAC_CTRL_1	0x0000	-N/A-	DAC Control 1
AD1836A_DAC_CTRL_2	0x1000	-N/A-	DAC Control 2
AD1836A_DAC_1L_VOL	0x2000	-N/A-	DAC 1L Volume
AD1836A_DAC_1R_VOL	0x3000	-N/A-	DAC 1R Volume
AD1836A_DAC_2L_VOL	0x4000	-N/A-	DAC 2L Volume
AD1836A_DAC_2R_VOL	0x5000	-N/A-	DAC 2R Volume
AD1836A_DAC_3L_VOL	0x6000	-N/A-	DAC 3L Volume
AD1836A_DAC_3R_VOL	0x7000	-N/A-	DAC 3R Volume
AD1836A_ADC_1L_VOL	0x8000	-N/A-	ADC 1L Peak Volume
AD1836A_ADC_1R_VOL	0x9000	-N/A-	ADC 1R Peak Volume
AD1836A_ADC_2L_VOL	0xA000	-N/A-	ADC 2L Peak Volume
AD1836A_ADC_2R_VOL	0xB000	-N/A-	ADC 2R Peak Volume
AD1836A_ADC_CTRL_1	0xC000	-N/A-	ADC Control 1
AD1836A_ADC_CTRL_2	0xD000	-N/A-	ADC Control 2
AD1836A_ADC_CTRL_3	0xE000	-N/A-	ADC Control 3

Table 7 – AD1836A device registers

7.2. AD1836A register fields

Field	Position	Size	Description
DAC Control 1 (AD1836A_DAC_CTRL_1)			
AD1836A_DE_EMPHASIS	8	2	De-Emphasis
AD1836A_SERIAL_MODE	5	3	Serial Mode
AD1836A_DAC_WORD_WIDTH	3	2	Data-Word width
AD1836A_DAC_PDN	2	1	Power-Down
AD1836A_INTERPOLATOR_MODE	1	1	Interpolator Mode
DAC Control 2 (AD1836A_DAC_CTRL_2)			
AD1836A_DAC_3R_MUTE	5	1	DAC 3R Mute
AD1836A_DAC_3L_MUTE	4	1	DAC 3L Mute

Field	Position	Size	Description
AD1836A_DAC_2R_MUTE	3	1	DAC 2R Mute
AD1836A_DAC_2L_MUTE	2	1	DAC 2L Mute
AD1836A_DAC_1R_MUTE	1	1	DAC 1R Mute
AD1836A_DAC_1L_MUTE	0	1	DAC 1L Mute
ADC Control 1 (AD1836A_ADC_CTRL_1)			
AD1836A_FILTER	8	1	Filter
AD1836A_ADC_PDN	7	1	ADC Power-Down
AD1836A_SAMPLE_RATE	6	1	Sample Rate
AD1836A_LEFT_GAIN	3	3	Left Gain
AD1836A_RIGHT_GAIN	0	3	Right Gain
ADC Control 2 (AD1836A_ADC_CTRL_2)			
AD1836A_AUX_MODE	9	1	Master/Slave AUX Mode
AD1836A_SOUT_MODE	6	3	SOUT Mode
AD1836A_ADC_WORD_WIDTH	4	2	Word Width
AD1836A_ADC_2R_MUTE	3	1	ADC 2R Mute
AD1836A_ADC_2L_MUTE	2	1	ADC 2L Mute
AD1836A_ADC_1R_MUTE	1	1	ADC 1R Mute
AD1836A_ADC_1L_MUTE	0	1	ADC 1L Mute
ADC Control 3 (AD1836A_ADC_CTRL_3)			
AD1836A_CLOCK_MODE	6	1	Clock Mode
AD1836A_LEFT_DIFF_SELECT	5	1	Left Differential I/P Select
AD1836A_RIGHT_DIFF_SELECT	4	1	Right Differential I/P Select
AD1836A_LEFT_MUX_ENABLE	3	1	Left MUX/PGA Enable
AD1836A_LEFT_MUX_SELECT	2	1	Left MUX I/P Select
AD1836A_RIGHT_MUX_ENABLE	1	1	Right MUX/PGA Enable
AD1836A_RIGHT_MUX_SELECT	0	1	Right MUX I/P Select

Table 8 – AD1836A Register Fields

8. Appendix

8.1. Using AD1836Aii Device Driver in Applications

This section explains how to use AD1836ii device driver in an application.

8.1.1. Device Manager Data memory allocation

This section explains device manager memory allocation requirements for applications using this driver. The application should allocate base memory + memory for one SPI device + memory for one SPORT device + memory for number of AD1836A device instances + memory for other devices used by the application

8.1.2. DMA Manager Data memory allocation

This section explains DMA manager memory allocation requirements for applications using this driver. The application should allocate base memory + memory for SPORT DMA channel(s) (1 DMA channel used for inbound or outbound data direction, 2 DMA channels for bidirectional data) + memory for DMA channels used by other devices in the application

Initialize Hardware (Ez-Kit), Interrupt manager, Deferred Callback Manager, DMA Manager, Device Manager (all application dependent)

8.1.3. Typical usage of AD1836A device driver

a. AD1836Aii (driver) initialization

Step 1: Open AD1836ii Device driver with device specific entry point (refer section 6.1 for valid entry point)

Step 2: Set SPORT device number to be used for AD1836A audio data flow
Example:
```
// Set AD1836A to use SPORT 0 for audio dataflow
adi_dev_Control (AD1836DriverHandle, ADI_AD1836A_CMD_SET_SPORT_DEVICE_NUMBER, (void *) 0);
```

Step 3: Open the above SPORT device that is to be used for AD1836A audio dataflow
Example:
```
// Open the SPORT device for audio dataflow
adi_dev_Control (AD1836ADriverHandle, ADI_AD1836A_CMD_SET_SPORT_STATUS,
                                    (void *) ADI_AD1836A_SPORT_OPEN);
```

b. AD1836A (hardware) initialization

Step 4: Set AD1836A SPI chip-select
Example:
```
// set Blackfin SPI chip-select for AD1836A is 6
adi_dev_Control(AD1836ADriverHandle, ADI_AD1836A_CMD_SET_SPI_CS,
                                (void *) 6);
```

Step 5: Configure AD1836A device to specific mode using device access service commands (refer section 8.2.2 for examples)

Optional steps (applicable only when application aims to configure SPORT device by its own or change SPORT settings by passing SPORT driver specific commands using AD1836A driver handle)

(Optional) Step 6: Disable 'auto-SPORT config' mode

Example:

```
// Disable AD1836A auto-SPORT config mode
adi_dev_Control(AD1836ADriverHandle, ADI_AD1836A_CMD_ENABLE_AUTO_SPORT_CONFIG,
                    (void *) FALSE);
```

(Optional) Step 7: Pass your own SPORT configuration table

Example:

```
// Pass SPORT configuration Table
adi_dev_Control(AD1836ADriverHandle, ADI_DEV_CMD_TABLE, (void *) Sport_Config_Table);
```

c. Audio Dataflow configuration

Step 8: Set audio dataflow method (this will be passed to SPORT driver)

Step 9: Load audio buffers for AD1836A device

Step 10: Enable AD1836A audio dataflow

d. Terminating AD1836A driver

Step11: Disable AD1836A audio dataflow

Step12: Terminate AD1836A driver with adi_dev_Terminate( )

Terminate DMA Manager, Deferred Callback etc.., (application dependent)

8.1.4. Re-use/share the SPORT device reserved by AD1836A

Application can reuse/share the same SPORT device reserved/presently used by AD1836A device, without closing the AD1836A driver itself.

```
// Close the SPORT device reserved by AD1836A device driver
adi_dev_Control (AD1836ADriverHandle, ADI_AD1836A_CMD_SET_SPORT_STATUS,
                    (void *) ADI_AD1836A_SPORT_CLOSE);
```

Application can also re-open the same/new SPORT device anytime by using the above command and AD1836A driver will configure the SPORT device in relevance to present AD1836A operating mode (provided the 'auto-SPORT config' mode is not disabled by the application)

8.1.5. Resetting AD1836A device

This driver has no control over AD1836A device reset pin and it is up to the application to reset AD1836A device before configuring it.

Application can use following lines of code to reset AD1836A device available on selected Blackfin Ez-Kits.

Resetting AD1836A device available on ADSP-BF533 Ez-Kit Lite:
AD1836A on ADSP- BF533 Ez-Kit – AD1836A reset pin is connected to flash port pins.

```
// Initialize flash memory on ADSP- BF533 Ez-Kit lite
// address of flash A port A output data register
#define pFlashA_PortA_Out   ((volatile unsigned char *) 0x20270004)
// address of flash A port A direction register
```

```
#define pFlashA_PortA_Dir   ((volatile unsigned char *) 0x20270006)
// Set Port A pin 0 as output
*pFlashA_PortA_Dir = 0x01;
// write 0 to Port A to reset AD1836
*pFlashA_PortA_Out = 0x00;
// wait at least 5 ns in reset
asm("nop; nop; nop; nop; nop;");
// write 1 to Port A to enable AD1836
*pFlashA_PortA_Out = 0x01;
```

Resetting AD1836A device available on ADSP-BF561 Ez-Kit Lite:
AD1836A on ADSP- BF561 Ez-Kit – AD1836A reset pin is connected to PF15 of BF561.

```
// Pull AD1836A from reset mode
// PF15 is connected to reset pin of AD1836A
adi_flag_Open(ADI_FLAG_PF15);
// set this flag pin as output
adi_flag_SetDirection(ADI_FLAG_PF15, ADI_FLAG_DIRECTION_OUTPUT);
// clear this pin output to reset AD1836A
adi_flag_Clear(ADI_FLAG_PF15);
//wait at least 5 ns in reset
asm("nop; nop; nop; nop; nop;");
// tie it to high again to enable AD1836A
adi_flag_Set(ADI_FLAG_PF15);
```

Resetting AD1836A device available on Blackfin A-V Ez-Extender:
AD1836A on Blackfin A-V Ez-Extender can be reset only via Ez-Kit Reset push button.

8.2. Accessing AD1836A registers

This section explains how to access the AD1836A internal registers using driver specific commands and device access commands (refer *'deviceaccess'* documentation for more information).

Refer section 7.1 for list of AD1838A device registers and section 7.2 for list of AD1838A device registers fields

8.2.1. Read AD1836A internal registers

1. Read a single register

```
// define the structure to access a single device register
ADI_DEV_ACCESS_REGISTER Read_Reg;

// Load the register address to be read
Read_Reg.Address = AD1836A_DAC_CTRL_1;

//clear the Data location
Read_Reg.Data = 0;
// Application calls adi_dev_Control( ) function with corresponding command and value
// Register value will be read back to location - Read_Reg.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_READ, (void *) & Read_Reg);
```

2. Read a specific register field

```
// define the structure to access a specific device register field
ADI_DEV_ACCESS_REGISTER_FIELD Read_Field;

// Load the device register address to be accessed
Read_Field.Address = AD1836A_ADC_CTRL_1;
// Load the device register field location to be read
Read_Field.Address = AD1836A_SAMPLE_RATE;

//clear the Read_Field.Data location
Read_Field.Data = 0;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The register field value will be read back to location - Read_Field.Data
adi_dev_Control (DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_READ, (void *) & Read_Field);
```

3. Read table of registers

```
// define the structure to access table of device registers
ADI_DEV_ACCESS_REGISTER Read_Regs [ ] = {
                                { AD1836A_DAC_CTRL_1,      0},
                                { AD1836A_ADC_CTRL_2,      0},
                                { AD1836A_ADC_CTRL_1,      0},
   /*MUST include delimiter */  {ADI_DEV_REGEND,   0}      // Register access delimiter
                                };

// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of registers listed above will be read to corresponding Data location in Read_Regs array
// i.e., value of AD1836A_DAC_CTRL_2 will be read to Read_Regs[0].Data,
// AD1836A_ADC_CTRL_2 to Read_Regs[1].Data and AD1836A_ADC_CTRL_1 to Read_Regs[2].Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_READ, (void *) &Read_Regs[0]);
```

4. Read table of register(s) fields

```
// define the structure to access table of device register(s) fields
ADI_DEV_ACCESS_REGISTER_FIELD Read_Fields [ ] = {
                                { AD1836A_ADC_CTRL_1,      AD1836A_ADC_PDN,          0},
                                { AD1836A_DAC_CTRL_2,      AD1836A_DAC_1L_MUTE,      0},
                                { AD1836A_ADC_CTRL_1,      AD1836A_SAMPLE_RATE,      0},
/*MUST include delimiter */ {ADI_DEV_REGEND,   0,         0}       // Register access delimiter
                                };

// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of register fields listed above will be read to corresponding Data location in Read_Fields array
// i.e., value of AD1836A_ADC_PDN will be read to Read_Fields[0].Data,
// AD1836A_DAC_1L_MUTE to Read_Fields [1].Data and AD1836A_SAMPLE_RATE to Read_Fields [2].Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_READ, (void *) & Read_Fields [0]);
```

5. Read block of registers

```
// define the structure to access a block of registers
ADI_DEV_ACCESS_REGISTER_BLOCK Read_Block;

// load the number of registers to be read
Read_Block.Count = 4;
// load the starting address of the register block to be read
Read_Block.Address = AD1836A_DAC_1L_VOL;
// define a 'Count' sized array to hold register data read from the device
u16 Block_Data[4] = { 0 };
// load the start address of the above array to Read_Block data pointer
Read_Block.pData = & Block_Data [0];

// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of the registers in the given block will be read to corresponding Block_Data[ ] array
// value of AD1836A_DAC_1L_VOL will be read to Block_Data [0],
// AD1836A_DAC_1R_VOL to Block_Data[1], AD1836A_DAC_2L_VOL to Block_Data[2]
// and AD1836A_DAC_2R_VOL to Block_Data[3]
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_BLOCK_READ, (void *) & Read_Block);
```

8.2.2. Configure AD1836A internal registers

1. Configure a single AD1836A register

```
// define the structure to access a single device register
ADI_DEV_ACCESS_REGISTER Cfg_Reg;

// Load the register address to be configured
Cfg_Reg.Address = AD1836A_DAC_1L_VOL;

//Load the configuration value to Cfg_Reg.Data location
Cfg_Reg.Data = 0x3FF;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The device register will be configured with the value in Cfg_Reg.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_WRITE, (void *) & Cfg_Reg);
```

2. Configure a specific register field

```
// define the structure to access a specific device register field
ADI_DEV_ACCESS_REGISTER_FIELD Cfg_Field;

// Load the device register address to be accessed
Cfg_Field.Address = AD1836A_ADC_CTRL_2;
// Load the device register field location to be configured
Cfg_Field.Address = AD1836A_AUX_MODE;

//load the new field value
Cfg_Field.Data = 1;
// Application calls adi_dev_Control( ) function with corresponding command and value
// Register field value will be read back to location - Cfg_Field.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_WRITE, (void *) & Cfg_Field);
```

3. Configure table of registers

```
// define the structure to access table of device registers (register address, register configuration value)
ADI_DEV_ACCESS_REGISTER Cfg_Regs [ ] = {
                                { AD1836A_DAC_CTRL_1,      0x000},
                                { AD1836A_DAC_CTRL_2,      0x000},
                                { AD1836A_DAC_1L_VOL,      0x3FF},
                                { AD1836A_DAC_1R_VOL,      0x3FF},
                                { AD1836A_DAC_CTRL_1,      0x000},
                                { AD1836A_DAC_CTRL_2,      0x180},
                                { AD1836A_DAC_CTRL_3,      0x000},
   /*MUST include delimiter */      {ADI_DEV_REGEND,   0    }   };       // Register access delimiter


// Application calls adi_dev_Control( ) function with corresponding command and value
// Registers listed in the table will be configured with corresponding table Data values
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) & Cfg_Regs [0]);
```

4. Configure a table of register(s) fields

```
// define the structure to access table of device register(s) fields
// register address, register field to configure, field configuration value
ADI_DEV_ACCESS_REGISTER_FIELD Cfg_Fields [ ] = {
                                { AD1836A_DAC_CTRL_1,      AD1836A_DAC_PDN,          0},
                                { AD1836A_DAC_CTRL_2,      AD1836A_DAC_1L_MUTE,      0},
                                { AD1836A_DAC_CTRL_2,      AD1836A_DAC_1L_MUTE,      0},
/*MUST include delimiter */ {ADI_DEV_REGEND,   0,         0}       // Register access delimiter
                                };

// Application calls adi_dev_Control( ) function with corresponding command and value
// Register fields listed in the above table will be configured with corresponding Data values
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) & Cfg_Fields [0]);
```

5. Configure a block of registers

```
// define the structure to access a block of registers
ADI_DEV_ACCESS_REGISTER_BLOCK Cfg_Block;

// load the number of registers to be configured
Cfg_Block.Count = 11;
// load the starting address of the register block to be configured
Cfg_Block.Address = AD1836A_DAC_CTRL_1;

// define a 'Count' sized array to hold register data read from the device
// load the array with AD1836A register configuration values
u16 Block_Cfg [11] = {0x000, 0x000, 0x3FF, 0x3FF, 0x3FF, 0x3FF, 0x3FF, 0x3FF, 0x000, 0x180, 0x000 };

// load the start address of the above array to Cfg_Block data pointer
Cfg_Block.pData = & Block_Cfg [0];

// Application calls adi_dev_Control( ) function with corresponding command and value
// Registers in the given block will be configured with corresponding values in Block_Cfg[ ] array
adi_dev_Control (DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) &Cfg_Block);
```