ANALOG DEVICES

ADI_ADAV801
DEVICE DRIVER

DATE: NOVEMBER 25, 2005.

Table of Contents

Document Revision History

Date	Description of Changes
2005/11/25	Initial release
2006/01/19	Compatibility to updated Device Access Service
2006/05/17	Updated to new device access interface Added register access examples

Table 1 – Revision History

1. Overview

The driver allows user to control ADAV801 DVD 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.

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_adav801.h > This file contains all definitions, function prototypes etc. specific to ADAV801 device

2.2. Source Files

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

- adi_adav801.c

3. Lower Level Drivers

ADAV801 driver is layered on SPI and SPORT drivers.

3.1. SPI Device Driver

ADAV801 can be operated in various modes by configuring its internal registers and it is done via SPI serial port.

3.2. SPORT Device Driver

Serial Port (SPORT) is used for audio data transfer to and from the codec. By default, ADAV801 device driver uses SPORT device 0 for its audio dataflow.

Application can directly communicate with the SPORT device allocated for ADAV801 audio dataflow by calling adi_dev_Control( ) function with PDDHandle specific to ADAV801 driver, command specific to the SPORT driver and value specific to the command

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 ADAV801 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. ADAV801 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 of Blackfin processor to audio I/O port pins of ADAV801.
Connect corresponding SPI device port pins of Blackfin processor to COUT, CIN, CCLK and CLATCH pins of ADAV801

Please 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_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_DATAFLOW
  - Enables/disables dataflow through the device
  - Value – TRUE/FALSE
- 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)

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_ADAV801_CMD_SET_SPI_CS
  - Sets SPI Chip-select for ADAV801
  - Value – u8
- ADI_ADAV801_CMD_GET_SPI_CS
  - Gets present SPI Chip-select used for ADAV801
  - Value – u8 *
- ADI_ADAV801_CMD_SET_SPORT_TX_WLEN
  - Sets SPORT transmit data word length
  - Value – u8
- ADI_ADAV801_CMD_SET_SPORT_RX_WLEN
  - Sets SPORT receive data word length
  - Value – u8
- ADI_ADAV801_CMD_SET_SPORT_DEVICE_NUMBER
  - Sets SPORT device number to be used for ADAV801 audio dataflow.
  - Value – u8
- ADI_ADAV801_CMD_SET_SPORT_STATUS
  - Sets status of SPORT device to be used for ADAV801 audio dataflow (Opens/Closes SPORT device)
  - Value – ADI_ADAV801_SET_SPORT_STATUS

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_TWI_LOCKED
  - Indicates the present TWI device is locked in other operation
- ADI_DEV_RESULT_REQUIRES_TWI_CONFIG_TABLE
  - Client need to supply a configuration table for the TWI driver
- 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_ADAV801_RESULT_CMD_NOT_SUPPORTED
  - Command supplied by the client is not supported by ADAV801 device driver
- ADI_ADAV801_RESULT_SPI_CS_NOT_VALID
  - SPI Chip-select supplied by the client is not valid
- ADI_ADAV801_RESULT_SPORT_WLEN_ILLEGAL
  - Results when the client provides an illegal SPORT Transmit/Receive word length
- ADI_ADAV801_RESULT_BAD_SPORT_DEVICE
  - Results when user provides a wrong SPORT device number
- ADI_ADAV801_RESULT_SPORT_STATUS_INVALID
  - Results when client tries to operate SPORT device in an invalid state

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.

- ADIADAV801EntryPoint

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 in the Blackfin processor	ADI_ADAV801_CMD_SET_SPORT_DEVICE_NUMBER
SPORT Transmit word length	24	between 2 and 31	ADI_ADAV801_CMD_SET_SPORT_TX_WLEN
SPORT Receive word length	24	between 2 and 31	ADI_ADAV801_CMD_SET_SPORT_RX_WLEN

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
SPORT status	ADI_ADAV801_SPORT_OPEN, ADI_ADAV801_SPORT_CLOSE	ADI_ADAV801_CMD_SET_SPORT_STATUS
SPI Chipselect	between 1 and 7	ADI_ADAV801_CMD_SET_SPI_CS

Table 5 – Additional Required Settings

7. Hardware Considerations

The client should set the SPI chip-select value (configure SPI_FLG) corresponding to ADAV801 before configuring the codec to specific mode. Command id 'ADI_ADAV801_CMD_SET_SPI_CS' can be used to set ADAV801's chip-select.

The following table is a list of registers that can be accessed on the ADAV801. Please refer to the ADAV801 device manual for a full description of registers and chip functionality.

Register	Address	Description
ADAV801_SRC_CLK_CTRL	0x00	SRC and Clock Control Register
ADAV801_SPDIF_CTRL	0x03	SPDIF Loopback Control Register
ADAV801_PBK_CTRL	0x04	Playback Port Control Register
ADAV801_AUXIN_PORT	0x05	Auxiliary Input Port Register
ADAV801_REC_CTRL	0x06	Record Port Control Register
ADAV801_AUXOUT_PORT	0x07	Auxiliary Output Port Register
ADAV801_GDLY_MUTE	0x08	Group Delay and Mute Register
ADAV801_RX_CON1	0x09	Receiver Configuration 1 Register
ADAV801_RX_CON2	0x0A	Receiver Configuration 2 Register
ADAV801_RXBUF_CON	0x0B	Receiver Buffer Configuration Register
ADAV801_TX_CTRL	0x0C	Transmitter Control Register
ADAV801_TXBUF_CON	0x0D	Transmitter Buffer Configuration Register
ADAV801_CSSBUF_TX	0x0E	Channel Status Switch Buffer and Transmitter
ADAV801_TXMSG_MSB	0x0F	Transmitter Message Zeros Most Significant Byte
ADAV801_TXMSG_LSB	0x10	Transmitter Message Zeros Least Significant Byte
ADAV801_AUTO_BUF	0x11	Autobuffer Register
ADAV801_SRR_MSB	0x12	Sample Rate Ratio MSB Register (Read Only
ADAV801_SRR_LSB	0x13	Sample Rate Ratio LSB Register (Read Only)
ADAV801_PBL_C_MSB	0x14	Preamble-C MSB Register (Read Only)
ADAV801_PBL_C_LSB	0x15	Preamble-C LSB Register (Read Only)
ADAV801_PBL_D_MSB	0x16	Preamble-D MSB Register (Read Only)
ADAV801_PBL_D_LSB	0x17	Preamble-D LSB Register (Read Only)
ADAV801_RX_ERR	0x18	Receiver Error Register (Read Only)
ADAV801_RX_ERR_MASK	0x19	Receiver Error Mask Register
ADAV801_SRC_ERR	0x1A	Sample Rate Conversion Error Register (Read Only)
ADAV801_SRC_ERR_MASK	0x1B	Sample Rate Conversion Error Mask Register
ADAV801_INT_STAT	0x1C	Interrupt Status Register
ADAV801_INT_STAT_MASK	0x1D	Interrupt Status Mask Register
ADAV801_MUTE_DEEMP	0x1E	Mute and De-Emphasis Register
ADAV801_NONAUDIO_PBL	0x1F	NonAudio Preamble Type Register (Read Only)
ADAV801_RXUSER_IADDR	0x50	Receiver User Bit Buffer Indirect Address Register
ADAV801_RXUSER_DATA	0x51	Receiver User Bit Buffer Data Register
ADAV801_TXUSER_IADDR	0x52	Transmitter User Bit Buffer Indirect Address Register
ADAV801_TXUSER_DATA	0x53	Transmitter User Bit Buffer Data Register
ADAV801_QCRC_ERR_STAT	0x54	Q Subcode CRCError Status Register (Read-Only)
ADAV801_Q_BUF1	0x55	Q Subcode Buffer 1 (Holds Address)
ADAV801_Q_BUF2	0x56	Q Subcode Buffer 2 (Holds Track Number)
ADAV801_Q_BUF3	0x57	Q Subcode Buffer 3 (Holds Index)
ADAV801_Q_BUF4	0x58	Q Subcode Buffer 4 (Holds Minute)
ADAV801_Q_BUF5	0x59	Q Subcode Buffer 5 (Holds Second)
ADAV801_Q_BUF6	0x5A	Q Subcode Buffer 6 (Holds Frame)
ADAV801_Q_BUF7	0x5B	Q Subcode Buffer 7 (Holds Zero)
ADAV801_Q_BUF8	0x5C	Q Subcode Buffer 8 (Holds Absolute Minute)
ADAV801_Q_BUF9	0x5D	Q Subcode Buffer 9 (Holds Absolute Second)
ADAV801_Q_BUF10	0x5E	Q Subcode Buffer 10 (Holds Absolute Frame)

Register	Address	Description
ADAV801_DP_CTRL1	0x62	Datapath Control Register 1
ADAV801_DP_CTRL2	0x63	Datapath Control Register 2
ADAV801_DAC_CTRL1	0x64	DAC Control Register 1
ADAV801_DAC_CTRL2	0x65	DAC Control Register 2
ADAV801_DAC_CTRL3	0x66	DAC Control Register 3
ADAV801_DAC_CTRL4	0x67	DAC Control Register 4
ADAV801_DACL_VOL	0x68	DAC Left Volume Register
ADAV801_DACR_VOL	0x69	DAC Right Volume Register
ADAV801_DACLPEAK_VOL	0x6A	DAC Left Peak Volume Register
ADAV801_DACRPEAK_VOL	0x6B	DAC Right Peak Volume Register
ADAV801_ADCLEFT_PGA	0x6C	ADC Left Channel PGA Gain Register
ADAV801_ADC_RIGHT_PGA	0x6D	ADC Right Channel PGA Gain Register
ADAV801_ADC_CTRL1	0x6E	ADC Control Register 1
ADAV801_ADC_CTRL2	0x6F	ADC Control Register 2
ADAV801_ADCL_VOL	0x70	ADC Left Volume Register
ADAV801_ADCR_VOL	0x71	ADC Right Volume Register
ADAV801_ADCLPEAK_VOL	0x72	ADC Left Peak Volume Register
ADAV801_ADCRPEAK_VOL	0x73	ADC Right Peak Volume Register
ADAV801_PLL_CTRL1	0x74	PLL Control Register 1
ADAV801_PLL_CTRL2	0x75	PLL Control Register 2
ADAV801_ICLK_CTRL1	0x76	Internal Clock Control Register 1
ADAV801_ICLK_CTRL2	0x77	Internal Clock Control Register 2
ADAV801_PLL_CLK_SRC	0x78	PLL Clock Source Register
ADAV801_PLLOUT_ENBL	0x7A	PLL Output Enable Register
ADAV801_ALC_CTRL1	0x7B	ALC Control Register 1
ADAV801_ALC_CTRL2	0x7C	ALC Control Register 2
ADAV801_ALC_CTRL3	0x7D	ALC Control Register 3

Table 6 – ADAV801 device registers

List of ADAV801 register fields.

Register Address: ADAV801_SRC_CLK_CTRL CTRL0
Register Fields:

- ADAV801_SRCDIV
- ADAV801_CLK2DIV
- ADAV801_CLK1DIV
- ADAV801_MCLKSEL

Register Address: ADAV801_SPDIF_LBK_CTRL CTRL0
Register Fields:

- ADAV801_TXMUX

Register Address: ADAV801_PBK_PORT_CTRL CTRL0
Register Fields:

- ADAV801_PBK_PORT_CLKSRC
- ADAV801_PBK_PORT_SPMODE

Register Address: ADAV801_AUX_INPUT_PORT CTRL0
Register Fields:

- ADAV801_AUX_IP_CLKSRC
- ADAV801_AUX_IP_SPMODE

Register Address: ADAV801_REC_PORT_CTRL CTRL0
Register Fields:
- ADAV801_REC_PORT_CLKSRC
- ADAV801_REC_PORT_WLEN
- ADAV801_REC_PORT_SPMODE

Register Address: ADAV801_AUX_OUTPUT_PORT CTRL0
Register Fields:
- ADAV801_AUX_OP_CLKSRC
- ADAV801_AUX_OP_WLEN
- ADAV801_AUX_OP_SPMODE

Register Address: ADAV801_GDLY_MUTE CTRL0
Register Fields:
- ADAV801_MUTE_SRC
- ADAV801_GRPDLY

Register Address: ADAV801_RX_CONFIG1 CTRL0
Register Fields:
- ADAV801_NOCLOCK
- ADAV801_RXCLK
- ADAV801_AUTO_DEEMPH
- ADAV801_RXERR
- ADAV801_RXLOCK

Register Address: ADAV801_RX_CONFIG2 CTRL0
Register Fields:
- ADAV801_RXMUTE
- ADAV801_SP_PLL
- ADAV801_SP_PLL_SEL
- ADAV801_NO_NONAUDIO
- ADAV801_NO_VALIDITY

Register Address: ADAV801_RXBUF_CONFIG CTRL0
Register Fields:
- ADAV801_RXBCONF5
- ADAV801_RXBCONF4
- ADAV801_RXBCONF3
- ADAV801_RXBCONF1
- ADAV801_RXBCONF0

Register Address: ADAV801_TX_CTRL CTRL0
Register Fields:
- ADAV801_TXVALIDITY
- ADAV801_TXRATIO
- ADAV801_TXCLKSEL
- ADAV801_TXENABLE

Register Address: ADAV801_TXBUF_CONFIG CTRL0
Register Fields:
- ADAV801_TX_IU_ZEROS
- ADAV801_TXBCONF3
- ADAV801_TXBCONF1
- ADAV801_TXBCONF0

Register Address: ADAV801_CSSBUF_TX CTRL0
Register Fields:
- ADAV801_TX_AB_SAME
- ADAV801_DISABLE_TX_COPY
- ADAV801_TXCSSWITCH
- ADAV801_RXCSSWITCH

Register Address: ADAV801_AUTO_BUF CTRL0
Register Fields:
- ADAV801_ZERO_STUFF_IU
- ADAV801_AUTO_UBITS
- ADAV801_AUTO_CSBITS
- ADAV801_AUTO_IU_ZEROS

Register Address: ADAV801_RX_ERROR CTRL0
Register Fields:
- ADAV801_RXERR_RXVALIDITY
- ADAV801_RXERR_EMPHASIS
- ADAV801_RXERR_NONAUDIO
- ADAV801_RXERR_NONAUDIO_PRE
- ADAV801_RXERR_CRC_ERROR
- ADAV801_RXERR_NOSTREAM
- ADAV801_RXERR_PARITY
- ADAV801_RXERR_LOCK

Register Address: ADAV801_RX_ERROR_MASK CTRL0
Register Fields:
- ADAV801_RXERR_RXVALIDITY_MASK
- ADAV801_RXERR_EMPHASIS_MASK
- ADAV801_RXERR_NONAUDIO_MASK
- ADAV801_RXERR_NONAUDIO_PRE_MASK
- ADAV801_RXERR_CRC_ERROR_MASK
- ADAV801_RXERR_NOSTREAM_MASK
- ADAV801_RXERR_PARITY_MASK
- ADAV801_RXERR_LOCK_MASK

Register Address: ADAV801_SRC_ERROR CTRL0
Register Fields:
- ADAV801_SRC_ERR_TOO_SLOW
- ADAV801_SRC_ERR_OVRL
- ADAV801_SRC_ERR_OVRR
- ADAV801_SRC_ERR_MUTE_IND

Register Address: ADAV801_SRC_ERROR_MASK CTRL0
Register Fields:
- ADAV801_SRC_ERR_OVRL_MASK
- ADAV801_SRC_ERR_OVRR_MASK
- ADAV801_SRC_ERR_MUTE_IND_MASK

Register Address: ADAV801_INT_STATUS CTRL0
Register Fields:
- ADAV801_SRCERROR
- ADAV801_TXCSTINT
- ADAV801_TXUBINT
- ADAV801_TXCSINT
- ADAV801_RXCSDIFF

- ADAV801_RXUBINT
- ADAV801_RXCSBINT
- ADAV801_RXERROR

Register Address: ADAV801_INT_STATUS_MASK CTRL0
Register Fields:
- ADAV801_MASK_SRC_ERROR
- ADAV801_MASK_TXCSTINT
- ADAV801_MASK_TXUBINT
- ADAV801_MASK_TXCSINT
- ADAV801_MASK_RXCSDIFF
- ADAV801_MASK_RXUBINT
- ADAV801_MASK_RXCSBINT
- ADAV801_MASK_RXERROR

Register Address: ADAV801_MUTE_DE_EMPHASIS CTRL0
Register Fields:
- ADAV801_TXMUTE
- ADAV801_SRC_DEEM

Register Address: ADAV801_NON_AUDIO_PREAMBLE CTRL0
Register Fields:
- ADAV801_DTS_CD_PBLE
- ADAV801_NONAUDIO_FRAME
- ADAV801_NONAUDIO_SUBFRAME_A
- ADAV801_NONAUDIO_SUBFRAME_B

Register Address: ADAV801_Q_CRC_ERROR_STATUS CTRL0
Register Fields:
- ADAV801_QCRCERROR
- ADAV801_QSUB

Register Address: ADAV801_DATAPATH_CTRL1 CTRL0
Register Fields:
- ADAV801_DATAPATH_SRC
- ADAV801_DATAPATH_REC
- ADAV801_DATAPATH_AUXO

Register Address: ADAV801_DATAPATH_CTRL2 CTRL0
Register Fields:
- ADAV801_DATAPATH_DAC
- ADAV801_DATAPATH_DIT

Register Address: ADAV801_DAC_CTRL1 CTRL0
Register Fields:
- ADAV801_DR_ALL
- ADAV801_DR_DIGITAL
- ADAV801_DAC_CHSEL
- ADAV801_DAC_POL
- ADAV801_DAC_MUTER
- ADAV801_DAC_MUTEL

Register Address: ADAV801_DAC_CTRL2 CTRL0
Register Fields:
- ADAV801_DMCLK
- ADAV801_DFS
- ADAV801_DEEM

Register Address: ADAV801_DAC_CTRL3 CTRL0
Register Fields:
- ADAV801_ZFVOL
- ADAV801_ZFDATA
- ADAV801_ZFPOL

Register Address: ADAV801_DAC_CTRL4 CTRL0
Register Fields:
- ADAV801_INTRPT
- ADAV801_ZEROSEL

Register Address: ADAV801_ADC_CTRL1 CTRL0
Register Fields:
- ADAV801_AMR
- ADAV801_HPF
- ADAV801_ADC_PDN
- ADAV801_ADC_ANA_PDN
- ADAV801_ADC_MUTER
- ADAV801_ADC_MUTEL
- ADAV801_PLPD
- ADAV801_PRPD

Register Address: ADAV801_ADC_CTRL2 CTRL0
Register Fields:
- ADAV801_BUF_PDN
- ADAV801_MCD

Register Address: ADAV801_PLL_CTRL1 CTRL0
Register Fields:
- ADAV801_DIRIN_CLK
- ADAV801_MCLKODIV
- ADAV801_PLLDIV
- ADAV801_PLL2PD
- ADAV801_PLL1PD
- ADAV801_XRLPD
- ADAV801_SYSCLK3

Register Address: ADAV801_PLL_CTRL2 CTRL0
Register Fields:
- ADAV801_PLL2_FS
- ADAV801_PLL2_SEL
- ADAV801_PLL2_DOUB
- ADAV801_PLL1_FS
- ADAV801_PLL1_SEL
- ADAV801_PLL1_DOUB

Register Address: ADAV801_ICLK_CTRL1 CTRL0
Register Fields:
- ADAV801_DCLK
- ADAV801_ACLK
- ADAV801_ICLK2

Register Address: ADAV801_ICLK_CTRL2 CTRL0
Register Fields:
- ADAV801_ICLK1
- ADAV801_PLL2INT
- ADAV801_PLL1INT

Register Address: ADAV801_PLL_CLK_SOURCE CTRL0
Register Fields:
- ADAV801_PLL1_SOURCE
- ADAV801_PLL2_SOURCE

Register Address: ADAV801_PLL_OUTPUT_ENBL CTRL0
Register Fields:
- ADAV801_DIRINPD
- ADAV801_DIRIN_PIN
- ADAV801_SYSCLK1_ENBL
- ADAV801_SYSCLK2_ENBL
- ADAV801_SYSCLK3_ENBL

Register Address: ADAV801_ALC_CTRL1 CTRL0
Register Fields:
- ADAV801_FSSEL
- ADAV801_GAINCNTR
- ADAV801_RECMODE
- ADAV801_LIMDET
- ADAV801_ALCEN

Register Address: ADAV801_ALC_CTRL2 CTRL0
Register Fields:
- ADAV801_RECTH
- ADAV801_ATKTH
- ADAV801_RECTIME
- ADAV801_ATKTIME

8. Appendix

8.1. Using ADAV801 Device Driver in Applications

This section explains how to use ADAV801 device driver in an application.

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 ADAV801 device + memory for other devices used by the application

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

a. ADAV801 (driver) initialization

Step 1: Open ADAV801 Device driver with device specific entry point (refer section 6.1 for valid entry points)

Step 2: Set SPORT device number to be used for ADAV801 audio data flow
Example:
```
// Set ADAV801 to use SPORT 0 for audio dataflow
adi_dev_Control (ADAV801DriverHandle, ADI_ADAV801_CMD_SET_SPORT_DEVICE_NUMBER,
                 (void *) 0);
```

b. ADAV801 (hardware) initialization

Step 4: Set ADAV801 SPI chip-select
Example:
```
// in this case, set Blackfin SPI chip-select for ADAV801 as 1
adi_dev_Control(ADAV801DriverHandle, ADI_ADAV801_CMD_SET_SPI_CS,
                (void *) 1);
```

Step 5: Configure ADAV801 device to specific mode using device access service (refer section 8.2.2 for examples)

c. Audio Dataflow configuration

Step 6: Open the above SPORT device via ADAV801 for audio dataflow
Example:
```
// Open the SPORT device for audio dataflow
adi_dev_Control (ADAV801DriverHandle, ADI_ADAV801_CMD_SET_SPORT_STATUS,
                 (void *) ADI_ADAV801_SPORT_OPEN);
```

Step 7: Set audio dataflow method

Step 8: Load ADAV801 audio buffers

Step 9: Enable ADAV801 audio dataflow

d. Terminating ADAV801 driver

Step10: Terminate ADAV801 driver with adi_dev_Terminate( )

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

8.2. Accessing ADAV801 registers

This section explains how to access the ADAV801 internal registers using driver specific commands and device access commands (refer *'deviceaccess'* documentation for more information).

Refer pages 17 and 18 for list of ADAV801 device registers and register fields

8.2.1. Read ADAV801 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 = ADAV801_SRC_CLK_CTRL;

//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 = ADAV801_PBK_CTRL;
// Load the device register field location to be read
Read_Field.Address = ADAV801_PBK_PORT_SPMODE;

//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 [ ] = {
                                { ADAV801_PLL_CTRL2,    0},
                                { ADAV801_DP_CTRL2,     0},
                                { ADAV801_PBK_CTRL,     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 ADAV801_PLL_CTRL2 will be read to Read_Regs[0].Data,
// ADAV801_DP_CTRL2 to Read_Regs[1].Data and ADAV801_PBK_CTRL 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 [ ] = {
                                { ADAV801_ADC_CTRL1,      ADAV801_AMR,              0},
                                { ADAV801_REC_CTRL,       ADAV801_REC_PORT_WLEN,    0},
                                { ADAV801_ADC_CTRL1,      ADAV801_ADC_PDN,          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 ADAV801_AMR will be read to Read_Fields[0].Data,
// ADAV801_REC_PORT_WLEN to Read_Fields [1].Data and ADAV801_ADC_PDN 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 = ADAV801_SRC_CLK_CTRL;
// 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 ADAV801_SRC_CLK_CTRL will be read to Block_Data [0],
// ADAV801_SPDIF_CTRL to Block_Data[1], ADAV801_PBK_CTRL to Block_Data[2]
// and ADAV801_AUXIN_PORT to Block_Data[3]
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_BLOCK_READ, (void *) & Read_Block);
```

8.2.2. Configure ADAV801 internal registers

1. Configure a single ADAV801 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 = ADAV801_PBK_CTRL;

//Load the configuration value to Cfg_Reg.Data location
Cfg_Reg.Data = 0x09;
// 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 = ADAV801_DP_CTRL2;
// Load the device register field location to be configured
Cfg_Field.Address = ADAV801_DATAPATH_DIT;

//load the new field value
Cfg_Field.Data = 2;
// 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 [ ] = {
                                { ADAV801_SRC_CLK_CTRL,         0x00},
                                { ADAV801_DP_CTRL2,             0x02},
                                { ADAV801_PBK_CTRL,             0x09},
                                { ADAV801_ADC_CTRL2,            0x10},
                                { ADAV801_PLL_CTRL2,            0xCC},
   /*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 [ ] = {
                                { ADAV801_PBK_CTRL,         ADAV801_PBK_PORT_CLKSRC,      1},
                                { ADAV801_DP_CTRL2,         ADAV801_DATAPATH_DIT,         2},
                                { ADAV801_PBK_CTRL,         ADAV801_PBK_PORT_SPMODE,      1},
/*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 = 10;
// load the starting address of the register block to be configured
Cfg_Block.Address = ADAV801_SRC_CLK_CTRL;

// define a 'Count' sized array to hold register data read from the device
// load the array with ADAV801 register configuration values
u16 Block_Cfg [10] = {0x00, 0x09, 0x00, 0x00, 0x00, 0x01, 0x00, 0x10, 0xCC,0x10};

// 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);
```