ANALOG DEVICES

ADI_SSM2602
DEVICE DRIVER

DATE: OCTOBER 05, 2007.

Table of Contents

List of Tables

Document Revision History

Date	Description of Changes
Oct 05 ,2007	Initial release

Table 1 – Revision History

1. Overview

Analog Devices' SSM2602 is a low power, high quality stereo audio codec for portable digital audio applications with stereo programmable gain amplifier (PGA) line and monaural microphone inputs. The document describes the functionality of SSM2602 driver that adheres to Analog Devices Device Driver and System Services Model.

Depending on selected serial interface format (2-wire or 3-wire), the driver leverages TWI or SPI driver to configure SSM2602 registers. The driver supports Device Access Commands to configure all or selected SSM2602 registers/register fields and to sense the present register/register field value (except the reset register). The driver maintains a register cache holding present value of SSM2602 registers (except the reset register) to support Device Access read commands.

The driver also leverages the SPORT driver to control the selected SPORT device and to handle the audio dataflow between Blackfin and SSM2602. SSM2602 driver also supports SPORT driver specific commands, which allows the application to configure/sense SPORT device using SSM2602 driver handle and SPORT specific commands.

Auto-SPORT Configuration support:

The driver can sense changes to SSM2602 Digital Audio Interface Format Register and automatically update the corresponding SPORT device configuration register to support the present interface mode, provided that the audio data format is in *I^2S or Left-Justified (LJ) Mode*. In other words, Auto-SPORT configuration is not supported for Right-Justified (RJ) and DSP modes. Application can enable/disable Auto-SPORT configuration support by issuing a driver specific command. Refer to section 5.7 for more information.

Auto-SPORT configuration support is enabled by default.

Support to WM8731/L codecs

ADI's SSM2602 driver can also be used to control/operate WM8731/WM8731L audio codecs.
Refer to sections 8.2 and 8.3 for more information.

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. specific to TWI or SPI Device Access.

- **<drivers/codec/adi_ssm2602.h>**
  This file contains all definitions, function prototypes etc. specific to SSM2602 device.

2.2. Source Files

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

- **<Blackfin/lib/src/drivers/codec/adi_ssm2602.c>**
  This file contains all the source code for the SSM2602 Device Driver. All source code is written in ‘C’. There are no assembly level functions in this driver.

3. Lower Level Drivers

SSM2602 driver is layered on TWI, SPI and SPORT drivers. Depending on the selected serial interface format (2-wire or 3-wire), the driver leverages TWI or SPI driver to configure SSM2602 device registers.

3.1. TWI Device Driver

SSM2602 can be operated in various modes by configuring its registers and can be done via TWI or SPI serial port. By default, SSM2602 device driver is set to use SPI device 0 to configure SSM2602 registers. The serial control interface mode is hardware dependent (status of SSM2602 MODE pin). Application can set the driver to use TWI to configure SSM2602 registers by issuing driver specific commands (refer section 5.4.3).

Note: It is Application's responsibility to set SSM2602 hardware in 2-wire interface mode. To use 2-wire serial interface, SSM2602 MODE pin must be set to 0.

3.2. SPI Device Driver

SSM2602 can be operated in various modes by configuring its registers and can be done via TWI or SPI serial port. By default, SSM2602 device driver is set to use SPI device 0 to configure SSM2602 registers. The serial control interface mode is hardware dependent (status of SSM2602 MODE pin). Application can set the driver to use SPI to configure SSM2602 registers by issuing driver specific commands (refer section 5.4.3).

Note: It is Application's responsibility to set SSM2602 hardware in 3-wire interface mode. To use 3-wire serial interface, SSM2602 MODE pin must be set to 1.

3.3. SPORT Device Driver

Serial Port (SPORT) is used to handle audio dataflow between Blackfin and the codec. By default, SSM2602 device driver is set to use SPORT device 0 for its audio dataflow.

Application can directly communicate with the SPORT driver to control the SPORT device allocated for SSM2602 audio dataflow using adi_dev_Control( ) function with SSM2602 driver Handle, SPORT driver specific command and command specific value. Application must open the SPORT device allocated to SSM2602 before issuing any SPORT specific command. Refer to section 5.4.3 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 SSM2602 device driver is build upon TWI driver, interrupt driven SPI driver and DMA operated SPORT driver.

4.1. Interrupts

SSM2602 driver requires two additional memory of size **`ADI_INT_SECONDARY_MEMORY`** for each (SPORT) DMA channel – one for DMA Data interrupt handler and one for DMA error interrupt handler. Additional memory of **`ADI_INT_SECONDARY_MEMORY`** size must be provided when the client decides to enable SPORT error reporting.

4.2. DMA

The driver doesn't support DMA directly, but uses a DMA driven SPORT for its audio dataflow. SSM2602 supports bi-directional dataflow and enough memory should be allocated for SPORT DMA channels depending on the codec's audio dataflow method.

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

Blackfin SPORT device port pins connected to Digital Audio Interface Port pins of SSM2602.
Blackfin TWI or SPI device port pins of connected to SSM2602 control interface.

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 that the SPORT device allocated for SSM2602 audio dataflow is opened before issuing any of these commands. Refer to SPORT driver documentation for details on SPORT driver 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.

SPI related commands

- **ADI_SSM2602_CMD_SET_SPI_DEVICE_NUMBER**
  - Sets SSM2602 driver to use SPI to access SSM2602 hardware registers
  - Value – u8 (SPI Device Number to use)
  - Default = 0 (SPI Device 0)
- **ADI_SSM2602_CMD_SET_SPI_CS**
  - Sets SPI Chip select number connected to SSM2602 CSB pin
  - Value – u8 (SPI Chip select)
  - Default = 0 (Chip select not valid)

TWI related commands

- **ADI_SSM2602_CMD_SET_TWI_DEVICE_NUMBER**
  - Sets SSM2602 driver to use TWI to access SSM2602 hardware registers
  - Value – u8 (TWI Device Number to use)
  - Default = uses SPI device 0
- **ADI_SSM2602_CMD_SET_TWI_GLOBAL_ADDR**
  - Sets SSM2602 TWI Global Address
  - Value – u8 (TWI Global Address for SSM2602)
  - Default = 0x1A (possible values = 0x1A or 0x1B)

Commands common to TWI and SPI

- **ADI_SSM2602_CMD_SET_TWI_SPI_CONFIG_TABLE**
  - Sets TWI or SPI configuration table to be used to access SSM2602 device registers
  - Value – ADI_DEV_CMD_VALUE_PAIR * (address of a command/value pair table)
  - Default = NULL
  - Note: Client MUST submit a configuration table when the driver is set to use TWI for SSM2602 register access. Configuration table is optional for SPI based SSM2602 register access

SPORT related commands

- **ADI_SSM2602_CMD_SET_SPORT_DEVICE_NUMBER**
  - Sets SPORT device number connected to SSM2602 Digital Audio Interface port
  - Value – u8 (SPORT Device Number to use)
  - Default = 0 (SPORT device 0)
- **ADI_SSM2602_CMD_OPEN_SPORT_DEVICE**
  - Opens/Closes SPORT device connected to SSM2602
  - Value – TRUE/FALSE (TRUE to open, FALSE to close)
  - Default = Opens SPORT device when application issues Set Dataflow method command
  - Note: Command can be used to share one SPORT channel between multiple Audio Codecs
- **ADI_SSM2602_CMD_ENABLE_AUTO_SPORT_CONFIG**
  - Enables/Disables Auto-SPORT configuration mode
  - Value – TRUE/FALSE (TRUE to enable, FALSE to disable)
  - Default = TRUE (Auto-SPORT configuration is enabled by default)
  - Note: Auto-SPORT configuration is not supported for Right-Justified (RJ) and DSP modes

SSM2602 Register configuration commands

- **`ADI_SSM2602_CMD_SET_SAMPLE_RATE`**
  - Configures SSM2602 sampling control register with given ADC/DAC sample rate. Application MUST disable codec dataflow before issuing this command.
  - Value – ADI_SSM2602_SAMPLE_RATE * (pointer to ADI_SSM2602_SAMPLE_RATE structure)
  - Note: It is applications' responsibility to set/clear Base over sampling (BOSR) bit in Sampling control register depending on the MCLK frequency.

```
/* Structure to pass SSM2602 ADC/DAC sampling rate */
typedef struct ADI_SSM2602_SAMPLE_RATE
{
        u32     AdcSampleRate;  /* ADC Sample rate  */
        u32     DacSampleRate;  /* DAC Sample rate  */
} ADI_SSM2602_SAMPLE_RATE;
```

- Use following Device Access Commands (defined in adi_dev.h) to access SSM2602 hardware registers. Refer section 5.4.2 for command arguments and section 8.5 for examples

Command ID	Comments
ADI_DEV_CMD_REGISTER_READ	Reads a single SSM2602 register
ADI_DEV_CMD_REGISTER_FIELD_READ	Reads a specific SSM2602 register field
ADI_DEV_CMD_REGISTER_TABLE_READ	Reads a table of SSM2602 registers
ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ	Reads a table of SSM2602 register(s) field(s)
ADI_DEV_CMD_REGISTER_BLOCK_READ	Reads a block of consecutive SSM2602 registers
ADI_DEV_CMD_REGISTER_WRITE	Writes to a single SSM2602 register
ADI_DEV_CMD_REGISTER_FIELD_WRITE	Writes to a SSM2602 register field
ADI_DEV_CMD_REGISTER_TABLE_WRITE	Writes to a table of SSM2602 registers
ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE	Writes to a table of SSM2602 register(s) field(s)
ADI_DEV_CMD_REGISTER_BLOCK_WRITE	Writes to a block of consecutive SSM2602 registers

Table 4 – Device Access Commands supported by SSM2602 driver

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_SSM2602_RESULT_CMD_NOT_SUPPORTED**
  - Command supplied by the client is not supported by SSM2602 device driver
- **ADI_SSM2602_RESULT_TWI_GLOBAL_ADDRESS_INVALID**
  - SSM2602 TWI Global address passed by the client is not valid.
- **ADI_SSM2602_RESULT_SAMPLE_RATE_NOT_SUPPORTED**
  - SSM2602 driver does not support the given ADC/DAC Sample rate.

5.7. Auto-SPORT Configuration

The driver can sense changes to SSM2602 Digital Audio Interface Format Register (usually carried out by the application) and automatically update the corresponding SPORT device configuration registers to support the present interface mode, provided that the audio data format is in *I^2S or Left-Justified (LJ) Mode*. In other words, Auto-SPORT configuration is not supported for Right-Justified (RJ) and DSP modes. Application can enable/disable Auto-SPORT configuration support by issuing driver specific command - **`ADI_SSM2602_CMD_ENABLE_AUTO_SPORT_CONFIG`** with command argument as TRUE/FALSE.

Please note that Auto-SPORT configuration only updates SPORT configuration registers (SPORT Transmit Control register 1 & 2, SPORT Receive control register 1 & 2). It is applications' responsibility to configure SPORT Transmit/Receive Clock divider and SPORT Transmit/Receive Frame Sync divider registers when the codec is operated in slave mode.

Auto-SPORT configuration support is enabled by default.

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.

- ADISSM2602EntryPoint

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
Auto-SPORT Configuration	TRUE	TRUE/FALSE	ADI_SSM2602_CMD_ENABLE_AUTO_SPORT_CONFIG

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 Device Number (provided that SSM2602 is operated in 3-wire Mode)	Depends on number of SPI devices available on the Blackfin processor (Default = 0)	ADI_SSM2602_CMD_SET_SPI_DEVICE_NUMBER
SPI Chipselect (provided that SSM2602 is operated in 3-wire Mode)	between 1 and 7	ADI_SSM2602_CMD_SET_SPI_CS
TWI Device Number (provided that SSM2602 is operated in 2-wire Mode)	Depends on number of TWI devices available on the Blackfin processor	ADI_SSM2602_CMD_SET_TWI_DEVICE_NUMBER
SSM2602 TWI Global Address (provided that SSM2602 is operated in 2-wire Mode)	0x1A or 0x1B (depends on SSM2602 CSB pin status)	ADI_SSM2602_CMD_SET_TWI_GLOBAL_ADDR
TWI Configuration Table (provided that SSM2602 is operated in 2-wire Mode)	ADI_DEV_CMD_VALUE_PAIR *	ADI_SSM2602_CMD_SET_TWI_SPI_CONFIG_TABLE
SPORT Device Number	Depends on number of SPORT devices available on the Blackfin processor (Default = 0)	ADI_SSM2602_CMD_SET_SPORT_DEVICE_NUMBER
Open/Close SPORT	TRUE/FALSE	ADI_SSM2602_CMD_OPEN_SPORT_DEVICE

Table 6 – Additional Required Settings

7. Hardware Considerations

When SSM2602 is operated in 2-wire mode, client must set the SSM2602 TWI Global Address (0x1A or 0x1B) Command id **ADI_SSM2602_CMD_SET_SPI_CS** can be used to set SSM2602's chip-select.

When SSM2602 is operated in 3-wire mode, client must set the SPI chip-select number used to select SSM2602. Command id **ADI_SSM2602_CMD_SET_TWI_GLOBAL_ADDR** can be used to set SSM2602's chip-select.

7.1. SSM2602 registers

The following table is a list of registers that can be accessed on the SSM2602. Refer to the SSM2602 hardware reference manual for register description and chip functionality.

Register	Address	Default	Description
SSM2602_REG_LEFT_ADC_VOL	0x00	0x004B	Left ADC Volume Control Register
SSM2602_REG_RIGHT_ADC_VOL	0x01	0x004B	Right ADC Volume Control Register
SSM2602_REG_LEFT_DAC_VOL	0x02	0x0079	Left DAC Volume Control Register
SSM2602_REG_RIGHT_DAC_VOL	0x03	0x0079	Right DAC Volume Control Register
SSM2602_REG_ANALOGUE_PATH	0x04	0x000A	Analogue Audio Path Control Register
SSM2602_REG_DIGITAL_PATH	0x05	0x0008	Digital Audio Path Control Register
SSM2602_REG_POWER	0x06	0x009F	Power Management Register
SSM2602_REG_DIGITAL_IFACE	0x07	0x000A	Digital Audio Interface Format Register
SSM2602_REG_SAMPLING_CTRL	0x08	0x0000	Sampling Control Register
SSM2602_REG_ACTIVE_CTRL	0x09	0x0000	Active Control Register
SSM2602_REG_RESET	0x0F	Non-zero	Reset Register (write only)
SSM2602_REG_ALC_1	0x10	0x007B	Automatic Level Control Register 1
SSM2602_REG_ALC_2	0x11	0x0032	Automatic Level Control Register 2
SSM2602_REG_NOISE_GATE	0x12	0x0000	Noise Gate Register

Table 7 – SSM2602 Device registers

7.2. SSM2602 register fields

Field	Mask	Size	Description
Left ADC Volume Control Register(SSM2602_REG_LEFT_ADC_VOL)			
SSM2602_RFLD_LIN_VOL	0x03F	6	Left Channel PGA Volume control
SSM2602_RFLD_LIN_ENABLE_MUTE	0x080	1	Left Channel Input Mute
SSM2602_RFLD_LRIN_BOTH	0x100	1	Left Channel Line Input Volume update
Right ADC Volume Control Register(SSM2602_REG_RIGHT_ADC_VOL)			
SSM2602_RFLD_RIN_VOL	0x03F	6	Right Channel Line Input volume control
SSM2602_RFLD_RIN_ENABLE_MUTE	0x080	1	Right Channel Input Mute
SSM2602_RFLD_RLIN_BOTH	0x100	1	Right Channel Line Input Volume update
Left DAC Volume Control Register (SSM2602_REG_LEFT_DAC_VOL)			
SSM2602_RFLD_LHP_VOL	0x07F	7	Left Channel Headphone volume control
SSM2602_RFLD_ENABLE_LZC	0x080	1	Left Channel Zero cross detect enable
SSM2602_RFLD_LRHP_BOTH	0x100	1	Left Channel Headphone volume update
Right DAC Volume Control Register (SSM2602_REG_RIGHT_DAC_VOL)			
SSM2602_RFLD_RHP_VOL	0x07F	7	Right Channel Headphone volume control
SSM2602_RFLD_ENABLE_RZC	0x080	1	Right Channel Zero cross detect enable
SSM2602_RFLD_RLHP_BOTH	0x100	1	Right Channel Headphone volume update
Analogue Audio Path Control Register (SSM2602_REG_ANALOGUE_PATH)			
SSM2602_RFLD_ENABLE_MIC_BOOST	0x001	1	Primary Microphone Amplifier gain booster control
SSM2602_RFLD_ENABLE_MIC_MUTE	0x002	1	Microphone Mute Control
SSM2602_RFLD_ADC_IN_SELECT	0x004	1	Microphone/Line IN select to ADC
SSM2602_RFLD_ENABLE_BYPASS	0x008	1	Line input bypass to line output
SSM2602_RFLD_SELECT_DAC	0x010	1	Select DAC (1=Select DAC, 0=Don't Select DAC)
SSM2602_RFLD_ENABLE_SIDETONE	0x020	1	Enable/Disable Side Tone
SSM2602_RFLD_SIDETONE_ATTN	0x0C0	2	Side Tone Attenuation
SSM2602_RFLD_ENABLE_MIC_BOOST_2	0x100	1	Additional Microphone amplifier gain booster control
Digital Audio Path Control Register (SSM2602_REG_DIGITAL_PATH)			
SSM2602_RFLD_ENABLE_ADC_HPF	0x001	1	Enable/Disable ADC High-pass Filter
SSM2602_RFLD_DE_EMPHASIS	0x006	2	De-Emphasis Control
SSM2602_RFLD_ENABLE_DAC_MUTE	0x008	1	DAC Mute Control
SSM2602_RFLD_HP_STORE_OFFSET	0x010	1	Store DC offset when HPF is disabled
Power Management Register (SSM2602_REG_POWER)			
SSM2602_RFLD_LINE_IN_PDN	0x001	1	Line Input Power Down
SSM2602_RFLD_MIC_PDN	0x002	1	Microphone Input & Bias Power Down
SSM2602_RFLD_ADC_PDN	0x004	1	ADC Power Down
SSM2602_RFLD_DAC_PDN	0x008	1	DAC Power Down
SSM2602_RFLD_OUT_PDN	0x010	1	Outputs Power Down
SSM2602_RFLD_OSC_PDN	0x020	1	Oscillator Power Down

Field	Mask	Size	Description
SSM2602_RFLD_CLK_OUT_PDN	0x040	1	CLKOUT Power Down
SSM2602_RFLD_POWER_OFF	0x080	1	POWEROFF Mode
Digital Audio Interface Format Register (SSM2602_REG_DIGITAL_IFACE)			
SSM2602_RFLD_IFACE_FORMAT	0x003	2	Digital Audio input format control
SSM2602_RFLD_AUDIO_DATA_LEN	0x00C	2	Input Audio Data Bit length select
SSM2602_RFLD_DAC_LR_POLARITY	0x010	1	Polarity Control for clocks in RJ,LJ and I2S modes
SSM2602_RFLD_DAC_LR_SWAP	0x020	1	Swap DAC data control
SSM2602_RFLD_ENABLE_MASTER	0x040	1	Enable/Disable Master Mode
SSM2602_RFLD_BCLK_INVERT	0x080	1	Bit Clock Inversion control
Sampling Control Register (SSM2602_REG_SAMPLING_CTRL)			
SSM2602_RFLD_ENABLE_USB_MODE	0x001	1	Enable/Disable USB Mode
SSM2602_RFLD_BOS_RATE	0x002	1	Base Over-Sampling rate
SSM2602_RFLD_SAMPLE_RATE	0x03C	4	Clock setting condition (Sampling rate control)
SSM2602_RFLD_CORECLK_DIV2	0x040	1	Core Clock divider select
SSM2602_RFLD_CLKOUT_DIV2	0x080	1	Clock Out divider select
Active Control Register(SSM2602_REG_ACTIVE_CTRL)			
SSM2602_RFLD_ACTIVATE_CODEC	0x001	1	Activate Codec Digital Audio Interface
Automatic Level Control Register 1 (SSM2602_REG_ALC_1)			
SSM2602_RFLD_ALC_TARGET	0x00F	4	Automatic Level Control Target level
SSM2602_RFLD_PGA_MAX_GAIN	0x070	3	Automatic Level Control - PGA maximum Gain
SSM2602_RFLD_SELECT_ALC	0x180	2	Automatic Level Control Selection
Automatic Level Control Register 2 (SSM2602_REG_ALC_2)			
SSM2602_RFLD_ALC_ATTACK_TIME	0x00F	4	Automatic Level Control - Attack time control
SSM2602_RFLD_ALC_DECAY_TIME	0x0F0	4	Automatic Level Control – Decay time control
Noise Gate Register (SSM2602_REG_NOISE_GATE)			
SSM2602_RFLD_ENABLE_NG	0x001	1	Enable/Disable Noise Gate
SSM2602_RFLD_NG_TYPE	0x006	2	Noise Gate Type
SSM2602_RFLD_NG_THRESHOLD	0x0F8	5	Noise Gate Threshold

Table 8 – SSM2602 Register Fields

8. Appendix

8.1. Updating Codec/SPORT registers

Application MUST disable the codec (SPORT) dataflow while performing any of the following operations.

- Configuring/updating SSM2602 Digital Audio Interface or Sampling Control register.
- Updating ADC/DAC sampling rate using `ADI_SSM2602_CMD_SET_SAMPLE_RATE` command.
- Configuring/updating SPORT registers.

8.2. Using ADI's SSM2602 driver with WM8731/L codec

SSM2602 driver can also be used to control/operate WM8731/L audio codecs, provided that the register/register field differences between SSM2602 and WM8731/L are carefully handled in the application. Table 9 lists the register/register field differences between the two codecs.

Register/Register Field	SSM2602	WM8731/L
Enable/Disable ADC High-pass filter (Bit 0 in Digital Audio Path Register)	0 to Disable, 1 to Enable ADC HPF disabled by default	1 to Disable, 0 to Enable ADC HPF enabled by default
Left Channel PGA Volume field in Left ADC Volume Control register	Bit [5:0] (6 bits) Range: +33dB to -34.5 dB	Bit [4:0] (5 bits) Range: +12dB to -34.5 dB
Right Channel PGA Volume field in Right ADC Volume Control register	Bit [5:0](6 bits) Range: +33dB to -34.5 dB	Bit [4:0] (5 bits) Range: +12dB to -34.5 dB
Additional Microphone amplifier gain booster (Bit 8 in Analogue Audio Path Register)	Available	Not Available
Automatic Level Control Register 1	Available	Not available
Automatic Level Control Register 2	Available	Not available
Noise Gate Register	Available	Not available
Sample rates supported (As listed in Sampling rate look-up table, Bit [5:2] in Sampling rate control register)	96kHz, 88.2kHz, 48kHz, 44.1kHz, 32kHz, 24kHz, 22kHz, 16kHz, 12kHz, 11kHz, 8.02kHz, 8kHz	96kHz, 88.2kHz, 48kHz, 44.1kHz, 32kHz, 8.02kHz, 8kHz

Table 9 – Differences between SSM2602 and WM8731/L

8.3. Supported Sampling rate combinations

List of ADC/DAC sampling rate combinations supported by SSM2602 hardware and the driver (as listed in SSM2602 codec data sheet - Sampling rate look-up table)

ADC Sampling Rate (in Hz)	DAC Sampling Rate (in Hz)
8000	8000
8020	8020
11000	11000
12000	12000
22000	22000
24000	24000
32000	32000
44100	44100
48000	48000
88200	88200
96000	96000
8000	48000
48000	8000
8020	44100
44100	8020

Table 10 – ADC/DAC Sampling rate combinations supported by SSM2602

List of ADC/DAC sampling rate combinations supported by WM8731/L hardware (as listed in WM8731 codec data sheet - Sampling rate look-up table)

ADC Sampling Rate (in Hz)	DAC Sampling Rate (in Hz)
8000	8000
8020	8020
32000	32000
44100	44100
48000	48000
88200	88200
96000	96000
8000	48000
48000	8000
8020	44100
44100	8020

Table 11 – ADC/DAC Sampling rate combinations supported by WM8731/L

SSM2602 or WM8731 can be operated in few other sampling rates which are not listed in Table 10 and Table 11

Example 1: To operate ADC and DAC at 4 kHz sampling rate
- Set ADC and DAC sampling rates to 8 kHz using `ADI_SSM2602_CMD_SET_SAMPLE_RATE` command
- Set CLKDIV2 field(`SSM2602_RFLD_CORECLK_DIV2`) in codec sampling control register to 1.

Example 2: To operate WM8731 ADC and DAC at 24 kHz sampling rate
- Set ADC and DAC sampling rates to 48 kHz using `ADI_SSM2602_CMD_SET_SAMPLE_RATE` command
- Set CLKDIV2 field(`SSM2602_RFLD_CORECLK_DIV2`) in codec sampling control register to 1.

8.4. Using SSM2602 Device Driver in Applications

This section explains how to use SSM2602 device driver in an application.

8.4.1. Interrupt Manager Data memory allocation

This section explains Interrupt manager memory allocation requirements for applications using this driver. Depending on the data direction, the application must allocate memory for SPORT DMA channels, where one DMA channel requires memory for two secondary interrupt handlers of size `ADI_INT_SECONDARY_MEMORY`. Additional memory of size `ADI_INT_SECONDARY_MEMORY` must be provided when the client decides to enable SPORT error reporting.

8.4.2. 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

8.4.3. 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 TWI/SPI device + memory for one SPORT device + memory for number of SSM2602 device instances + memory for other devices used by the application

8.4.4. Typical usage of SSM2602 device driver in TWI Mode

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

a. SSM2602 (driver) initialization

Step 1: Open SSM2602 Device driver with device specific entry point (refer section 6.1 for valid entry point)

Step 2: Set TWI device number to be used for SSM2602 to access SSM2602 hardware registers

```
/* Example: Set SSM2602 to use to use TWI 0 to access SSM2602 hardware registers */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_SET_TWI_DEVICE_NUMBER, (void *) 0);
```

Step 3: Set SSM2602 TWI Global Address

```
/* Example: Set SSM2602 TWI Global Address */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_SET_TWI_GLOBAL_ADDR, (void *) 0x1A);
```

Step 4: Pass TWI Configuration Table

```
/* Example: Pass TWI Global Address */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_SET_TWI_SPI_CONFIG_TABLE,
                                                (void *)TWI_Config_Table);
```

Step 5: Set SPORT device number connected to SSM2602 Digital Audio Interface port (used for audio dataflow)

```
/* Example: Set SSM2602 to use SPORT 0 for audio dataflow */
adi_dev_Control (SSM2602DriverHandle,ADI_SSM2602_CMD_SET_SPORT_DEVICE_NUMBER,(void *) 0);
```

(Optional) Step 6: Open the above SPORT device that is to be used for SSM2602 audio dataflow

```
/* Example: Open the SPORT device for audio dataflow */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_OPEN_SPORT_DEVICE,(void *) TRUE);
```

b. SSM2602 (hardware) initialization

Step 7: Configure SSM2602 device to specific mode using device access service commands (refer section 8.5.2 for examples)

Step 8: Set ADC/DAC Sampling rates (refer to section 8.3 for list of supported sampling rates)

Example:

```
/* Data structure to pass sampling rate information */
ADI_SSM2602_SAMPLE_RATE     SampleRate;
/* Set ADC Sampling rate */
SampleRate.AdcSampleRate = 48000;
/* Set DAC Sampling rate */
SampleRate.DacSampleRate = 48000;
/* Pass ADC/DAC Sampling rate */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_SET_SAMPLE_RATE,(void *)&SampleRate);
```

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 SSM2602 driver handle)

(Optional) Step 9: Disable 'Auto-SPORT configuration' support

```
/* Example: Disable SSM2602 auto-SPORT configuration support */
adi_dev_Control(SSM2602DriverHandle, ADI_SSM2602_CMD_ENABLE_AUTO_SPORT_CONFIG, (void *) FALSE);
```

(Optional) Step 10: Pass your own SPORT configuration table

```
/* Example:  Pass SPORT configuration Table */
adi_dev_Control(SSM2602DriverHandle, ADI_DEV_CMD_TABLE, (void *) Sport_Config_Table);
```

c. Audio Dataflow configuration

Step 11: Set audio dataflow method (this will be passed to SPORT driver)

```
/* Example: Set SSM2602 in circular buffer mode */
adi_dev_Control(SSM2602DriverHandle, ADI_DEV_CMD_SET_DATAFLOW_METHOD,
                                                    (void *)ADI_DEV_MODE_CIRCULAR);
```

Step 12: Submit audio buffers

```
/* Example: Submit Inbound Audio Buffer */
adi_dev_Read(SSM2602DriverHandle, ADI_DEV_CIRC, (ADI_DEV_BUFFER *)&InboundBuffer);
/* Example:  Submit Outbound Audio Buffer */
adi_dev_Write(SSM2602DriverHandle, ADI_DEV_CIRC, (ADI_DEV_BUFFER *)&OutboundBuffer);
```

Step 13: Enable SSM2602 audio dataflow

```
/* Example:  Enable SSM2602 Audio Dataflow */
adi_dev_Control(SSM2602DriverHandle, ADI_DEV_CMD_SET_DATAFLOW, (void *)TRUE);
```

d. Terminating SSM2602 driver

Step14: Disable SSM2602 audio dataflow

```
/* Example: Disable SSM2602 Audio Dataflow */
adi_dev_Control(SSM2602DriverHandle, ADI_DEV_CMD_SET_DATAFLOW, (void *)FALSE);
```

Step15: Close SSM2602 driver

```
/* Example: Close SSM2602 driver */
adi_dev_Close(SSM2602DriverHandle);
```

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

8.4.5. Typical usage of SSM2602 device driver in SPI Mode

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

a. SSM2602 (driver) initialization

Step 1: Open SSM2602 Device driver with device specific entry point (refer section 6.1 for valid entry point)

Step 2: Set SPI device number to be used for SSM2602 to access SSM2602 hardware registers

```
/* Example: Set SSM2602 to use to use SPI 0 to access SSM2602 hardware registers */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_SET_SPI_DEVICE_NUMBER, (void *) 0);
```

Step 3: Set SPI Chip select connected to SSM2602

```
/* Example: Set SSM2602 to use to use SPI Chip select 5 to select SSM2602 */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_SET_SPI_CS, (void *) 5);
```

Step 4: Set SPORT device number connected to SSM2602 Digital Audio Interface port (used for audio dataflow)

```
/* Example: Set SSM2602 to use SPORT 0 for audio dataflow */
adi_dev_Control (SSM2602DriverHandle,ADI_SSM2602_CMD_SET_SPORT_DEVICE_NUMBER,(void *) 0);
```

(Optional) Step 5: Open the above SPORT device that is to be used for SSM2602 audio dataflow

```
/* Example: Open the SPORT device for audio dataflow */
adi_dev_Control (SSM2602DriverHandle,ADI_SSM2602_CMD_OPEN_SPORT_DEVICE,(void *) TRUE);
```

b. SSM2602 (hardware) initialization

Step 6: Configure SSM2602 device to specific mode using device access service commands (refer section 8.5.2 for examples)

Step 7: Set ADC/DAC Sampling rates (refer to section 8.3 for list of supported sampling rates)

Example:

```
/* Data structure to pass sampling rate information */
ADI_SSM2602_SAMPLE_RATE     SampleRate;
/* Set ADC Sampling rate */
SampleRate.AdcSampleRate = 48000;
/* Set DAC Sampling rate */
SampleRate.DacSampleRate = 48000;
/* Pass ADC/DAC Sampling rate */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_SET_SAMPLE_RATE,(void *)&SampleRate);
```

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 SSM2602 driver handle)

(Optional) Step 8: Disable 'Auto-SPORT configuration' support

```
/* Example: Disable SSM2602 auto-SPORT configuration support */
adi_dev_Control(SSM2602DriverHandle, ADI_SSM2602_CMD_ENABLE_AUTO_SPORT_CONFIG, (void *) FALSE);
```

(Optional) Step 9: Pass your own SPORT configuration table

```
/* Example: Pass SPORT configuration Table */
adi_dev_Control(SSM2602DriverHandle, ADI_DEV_CMD_TABLE, (void *) Sport_Config_Table);
```

c. Audio Dataflow configuration

Step 10: Set audio dataflow method (this will be passed to SPORT driver)

```
/* Example: Set SSM2602 in circular buffer mode */
adi_dev_Control(SSM2602DriverHandle, ADI_DEV_CMD_SET_DATAFLOW_METHOD,
                                                    (void *)ADI_DEV_MODE_CIRCULAR);
```

Step 11: Submit audio buffers

```
/* Example: Submit Inbound Audio Buffer */
adi_dev_Read(SSM2602DriverHandle, ADI_DEV_CIRC, (ADI_DEV_BUFFER *)&InboundBuffer);
// Example: Submit Outbound Audio Buffer
adi_dev_Write(SSM2602DriverHandle, ADI_DEV_CIRC, (ADI_DEV_BUFFER *)&OutboundBuffer);
```

Step 12: Enable SSM2602 audio dataflow

```
/* Example: Enable SSM2602 Audio Dataflow */
adi_dev_Control(SSM2602DriverHandle, ADI_DEV_CMD_SET_DATAFLOW, (void *)TRUE);
```

d. Terminating SSM2602 driver

Step13: Disable SSM2602 audio dataflow

```
/* Example: Disable SSM2602 Audio Dataflow */
adi_dev_Control(SSM2602DriverHandle, ADI_DEV_CMD_SET_DATAFLOW, (void *)FALSE);
```

Step14: Close SSM2602 driver

```
/* Example: Close SSM2602 driver */
adi_dev_Close(SSM2602DriverHandle);
```

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

8.4.6. Re-use/share the SPORT device reserved by SSM2602

Application can reuse/share the same SPORT device reserved used by SSM2602 device, without closing the SSM2602 driver itself.

```
/* Example: Close the SPORT device reserved by SSM2602 device driver */
adi_dev_Control (SSM2602DriverHandle, ADI_SSM2602_CMD_OPEN_SPORT_DEVICE, (void *) FALSE);
```

Application can also re-open the same/new SPORT device anytime by using the above command and SSM2602 driver will configure the SPORT device in relevance to present SSM2602 operating mode (provided the 'Auto-SPORT configuration' support is enabled)

8.4.7. Resetting codec registers

The audio codec registers can be reset to default values by writing zero to codec reset register.

8.5. Accessing SSM2602 registers

This section explains how to access the SSM2602 internal registers using device access commands (refer *'adi_deviceaccess'* documentation for more information).

Refer section 7.1 for list of SSM2602 device registers and section 7.2 for list of SSM2602 device registers fields

8.5.1. Read SSM2602 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 = SSM2602_REG_LEFT_ADC_VOL;

/* 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 = SSM2602_REG_LEFT_ADC_VOL;
/* Load the device register field location to be read */
Read_Field.Address = SSM2602_RFLD_LIN_VOL;

/* clear the Read_Field.Data location */
Read_Field.Data = 0;
/* Application calls adi_dev_Control( ) function with corresponding command and value */
/* 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 [ ] = {
                                        { SSM2602_REG_LEFT_DAC_VOL,     0},
                                        { SSM2602_REG_RIGHT_DAC_VOL,    0},
                                        { SSM2602_REG_DIGITAL_IFACE,    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 SSM2602_REG_LEFT_DAC_VOL will be read to Read_Regs[0].Data,
  SSM2602_REG_RIGHT_DAC_VOL to Read_Regs[1].Data and
  SSM2602_REG_DIGITAL_IFACE 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 [ ] = {
                        { SSM2602_REG_DIGITAL_IFACE,    SSM2602_RFLD_IFACE_FORMAT,      0},
                        { SSM2602_REG_DIGITAL_IFACE,    SSM2602_RFLD_AUDIO_DATA_LEN,    0},
                        { SSM2602_REG_SAMPLING_CTRL,    SSM2602_RFLD_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 SSM2602_RFLD_IFACE_FORMAT will be read to Read_Fields[0].Data,
   SSM2602_RFLD_AUDIO_DATA_LEN to Read_Fields [1].Data and
   SSM2602_RFLD_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 = SSM2602_REG_LEFT_ADC_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 SSM2602_REG_LEFT_ADC_VOL will be read to Block_Data [0],
  SSM2602_REG_RIGHT_ADC_VOL to Block_Data[1], SSM2602_REG_LEFT_DAC_VOL to Block_Data[2]
  and SSM2602_REG_RIGHT_DAC_VOL to Block_Data[3] */
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_BLOCK_READ, (void *) &Read_Block);
```

8.5.2. Configure SSM2602 internal registers

1. Configure a single SSM2602 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 = SSM2602_REG_RIGHT_ADC_VOL;

/* Load the configuration value to Cfg_Reg.Data location */
Cfg_Reg.Data = 0x1F;
/* 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 = SSM2602_REG_LEFT_ADC_VOL;
/* Load the device register field location to be configured */
Cfg_Field.Address = SSM2602_RFLD_LIN_VOL;

/* load the new field value */
Cfg_Field.Data = 0x18;
/* Application calls adi_dev_Control( ) function with corresponding command and value */
/* Register field will be updated to 'Cfg_Field.Data' value */
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 [ ] = {
                        { SSM2602_REG_LEFT_ADC_VOL,     0x17    },
                        { SSM2602_REG_RIGHT_ADC_VOL,    0x17    },
                        { SSM2602_REG_LEFT_DAC_VOL,     0x79    },
                        { SSM2602_REG_RIGHT_DAC_VOL,    0x79    },
                        { SSM2602_REG_POWER,            0x00    },
/*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 [ ] = {
    { SSM2602_REG_DIGITAL_IFACE,     SSM2602_RFLD_ENABLE_MASTER,          1},
    { SSM2602_REG_SAMPLING_CTRL,     SSM2602_RFLD_ENABLE_USB_MODE,        0},
    { SSM2602_REG_SAMPLING_CTRL,     SSM2602_RFLD_BOS_RATE,               0},
    { SSM2602_REG_SAMPLING_CTRL,     SSM2602_RFLD_SAMPLE_RATE,            3},
    { SSM2602_REG_ACTIVE_CTRL,       SSM2602_RFLD_ACTIVATE_CODEC,         1},
    {ADI_DEV_REGEND,                 0, 0}     /* Register access delimiter (MUST include 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 = 4;
/* load the starting address of the register block to be configured */
Cfg_Block.Address = SSM2602_REG_LEFT_ADC_VOL;

/* define a 'Count' sized array to hold register data read from the device */
/* load the array with SSM2602 register configuration values */
u16 Block_Cfg[4]    = {0x017, 0x017, 0x079, 0x079 };

/* 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);
```

9. References

1. Analog Devices SSM2602 Low Power Audio Codec DataSheet, Rev PrB, Sep 2007
http://www.analog.com/UploadedFiles/Data_Sheets/SSM2602.pdf