ADI_BF506FADC1 DEVICE DRIVER

DATE: DECEMBER 22, 2009.

Table of Contents

List of Tables

Document Revision History

Date	Description of Changes
Dec 22 ,2009	Initial release

Table 1 – Revision History

1. Overview

Analog Devices ADSP-BF506F Blackfin processor consists of an internal ADC. It is a dual, 12-bit, high speed, low power, successive approximation ADC that operates from a single 2.7 V to 5.25 V power supply and features throughput rates up to 2 MSPS. This document describes the functionality of the BF506F ADC driver that adheres to Analog Devices Device Driver and System Services Model.

BF506F ADC driver allows programmer to choose between ACM or GPIO pins to control the ADC operation. ADC data is received over a SPORT channel and programmer can use a driver control command to select a specific data mode based on the required number of channels.

BF506F ADC is built on top of the SPORT driver and ACM service. The driver leverages the SPORT driver to control the selected SPORT device and to handle the audio dataflow between the ADC and BF506F processor. Apart from ADC driver specific commands defined in section 5.4.3, the driver allows programmer to issue ACM service specific and SPORT driver specific commands to configure the respective peripherals.

Depending on the data mode selected by the application, the driver automatically configures the SPORT device registers to specific word length and enable/disable secondary receive channel. Programmer can choose between ACM or GPIO flag pins to control ADC. By default, the driver is set to use ACM. Using driver specific commands, application can configure the driver to switch between GPIO or ACM control. When the driver is set to use GPIO to control ADC, then the SPORT device will be configured for specified sampling rate with clock signals generated internally by SPORT.

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.

- **`<services/acm/adi_acm.h>`**
  This file contains all definitions, function prototypes etc. specific to ACM.

- **`<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/adc/adi_bf506fadc1.h>`**
  This file contains all definitions, function prototypes etc. specific to ADSP-BF560F ADC.

2.2. Source Files

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

- **`<Blackfin/lib/src/drivers/adc/adi_bf506fadc1.c>`**
  This file contains all the source code for the ADSP-BF506F ADC Device Driver. All source code is written in ‘C’. There are no assembly level functions in this driver

3. Lower Level Drivers

ADSP-BF506F ADC driver is layered on ACM service and SPORT driver. Depending on the selected data mode, the driver leverages SPORT driver to configure SPORT peripheral registers.

3.1. ACM Service

Application can use BF506F ADC driver handle to issue ACM specific commands, provided that the driver is set to use ACM to control ADC and the ACM device is already open. Please refer to ACM service manual for ACM specific commands and API.

3.2. SPORT Device Driver

Serial Port (SPORT) is used to receive data sample from ADC to Blackfin. By default, BF506F ADC device driver is set to use SPORT device 0 as its dataflow channel.

Application can directly communicate with the SPORT driver and set/sense current settings of SPORT device allocated for BF506F ADC using **`adi_dev_Control()`** function. Application can use BF506F ADC driver handle to issue SPORT driver specific commands. Application must open the SPORT device allocated to ADC before issuing any SPORT specific commands. Please refer to SPORT driver manual for SPORT specific commands.

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 BF506F ADC device driver is build upon ACM Service and DMA operated SPORT driver.

4.1. Interrupts

BF506F ADC driver requires two additional memory of size **`ADI_INT_SECONDARY_MEMORY`** for SPORT Rx DMA channel – one for Rx DMA Data interrupt handler and one for Rx 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 dataflow. BF506F ADC only supports inbound dataflow and enough memory should be allocated to support a SPORT DMA channel.

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

Application can use BF506F ADC driver specific commands to select between ACM and programmable Flags (GPIO) to control ADC.

4.6. Pins

Blackfin SPORT device port pins connected to data port of BF506F ADC.
Blackfin ACM peripheral port pins or GPIO pins connected to BF506F ADC 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.

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)

This driver also supports following commands and all SPORT specific commands, provided that the SPORT device allocated to manage BF506F ADC 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.

- **ADI_BF506ADC1_CMD_SET_ACM_DEV_NUMBER**
  - Sets ACM Device number to be used to control ADC
  - Value – **u8** (ACM Device Number)
  - Default = 0 (ACM Device 0)
- **ADI_BF506ADC1_CMD_SET_SPORT_DEV_NUMBER**
  - Sets SPORT Device number connected to ADC
  - Value – **u8** (SPORT Device Number)
  - Default = 0 (SPORT Device 0)
- **ADI_BF506ADC1_CMD_SET_ADC_CTRL_FLAG_PINS**
  - Sets Flag IDs of all ADC control signals and configures the driver to control ADC via GPIO flag pins
  - Value – address to structure of type **ADI_BF506FADC1_CTRL_FLAG_PINS**
  - Default = No Flag pins, uses ACM to control ADC
- **ADI_BF506ADC1_CMD_OPEN_ADC_CTRL_DEV**
  - Opens/Closes the device used to control ADC (ACM or GPIO Flags)
  - Value – **true/false** ('true' to open, 'false' to close)
  - Default = **false** (Control device closed)
- **ADI_BF506ADC1_CMD_OPEN_SPORT_DEV**
  - Opens/Closes SPORT device connected to ADC
  - Value – **true/false** ('true' to open, 'false' to close)
  - Default = **false** (SPORT device closed)
- **ADI_BF506ADC1_CMD_SET_ADC_CTRL_FLAG_STATE**
  - Sets the value/state of ADC control signals when the driver is set to control ADC via GPIO flag pins
  - Value – address to structure of type **ADI_BF506FADC1_CTRL_FLAG_STATE**
  - Default = No Flag pins, uses ACM to control ADC
- **ADI_BF506ADC1_CMD_SET_SERIAL_DATA_MODE**
  - Sets Serial data mode to receive ADC data
  - Value – Enumeration of type **ADI_BF506FADC1_DATA_MODE**
  - Default = **ADI_BF506FADC1_DATA_DOUT_A_ONLY**
- **ADI_BF506ADC1_CMD_SET_OPERATING_MODE**
  - Sets ADC Operating mode
  - Value – enumeration of type **ADI_BF506ADC1_OP_MODE**
  - Default = **ADI_BF506ADC1_MODE_POWERDOWN**
- **ADI_BF506ADC1_CMD_SET_SAMPLE_RATE_ACLK_FREQ**
  - Sets ADC sampling rate / ACLK frequency
  - Value – **u32** (Sampling rate / ACLK freq in Hertz)
  - Default = 20000000 (20MHz)
  - Note: The value will be considered as
    - ADC sample rate when driver is configured to use GPIO flag pins to control ADC
    - ACLK Frequency when driver is configured to use ACM to control ADC

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 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 dataflow method, this value is the **`CallbackParameter`** value that was supplied in the buffer that was passed to the **`adi_dev_Read()`** function. For the circular dataflow method, this value is the address of the buffer provided in the **`adi_dev_Read()`**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()`**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_WRITE_ON_INBOUND_DEVICE`**
  - The client attempted to provide an outbound buffer for a device opened for inbound traffic only.
- **`ADI_DEV_RESULT_DATAFLOW_UNDEFINED`**
  - The dataflow method has not yet been declared.
- **`ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE`**
  - The dataflow method is incompatible with the action requested.
- **`ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE`**
  - The device does not support the buffer type provided.

- **ADI_DEV_RESULT_CANT_HOOK_INTERRUPT**
  - The Interrupt Manager failed to hook an interrupt handler.
- **ADI_DEV_RESULT_CANT_UNHOOK_INTERRUPT**
  - The Interrupt Manager failed to unhook an interrupt handler.
- **ADI_DEV_RESULT_NON_TERMINATED_LIST**
  - The chain of buffers provided is not NULL terminated.
- **ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED**
  - No callback function was supplied when it was required.
- **ADI_DEV_RESULT_REQUIRES_UNIDIRECTIONAL_DEVICE**
  - Requires the device be opened for either inbound or outbound traffic only.
- **ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE**
  - Requires the device be opened for bidirectional traffic only.

5.6.2. Device Driver Specific Return Codes

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

- **ADI_BF506ADC1_RESULT_CMD_NOT_SUPPORTED**
  - Command supplied by the client is not supported by BF506F ADC device driver
- **ADI_BF506ADC1_RESULT_CANNOT_PROCESS_CMD**
  - Results when the issued command is supported, but can not be processed at current stage
- **ADI_BF506ADC1_RESULT_DATA_MODE_INVALID**
  - Results when the issued Data mode is invalid.
- **ADI_BF506ADC1_RESULT_OPERATING_MODE_INVALID**
  - Results when the issued Operating mode is invalid.
- **ADI_BF506ADC1_RESULT_SAMPLE_RATE_ACLK_INVALID**
  - Results when the issued Sample Rate / ACLK Freq is invalid.
- **ADI_BF506ADC1_RESULT_SAMPLE_RATE_ACLK_NOT_SUPPORTED**
  - Results when the issued Sample Rate / ACLK Freq is not supported by current SCLK.

5.7. Enumerations

5.7.1. Enumerations of ADC data modes supported by the driver

```
typedef enum __AdiBf506fadc1DataMode
{
    /*
    ** Note that regardless of the mode, the first two bits of
    ** each ADC sample will always be zero
    */

    /* Receive Data only from Data port A with no extended zeros (14-bits) */
    ADI_BF506FADC1_DATA_DOUT_A_ONLY = 0,
    /* Receive Data only from Data port A with extended zeros (16-bits) */
    ADI_BF506FADC1_DATA_DOUT_A_ONLY_XTND_ZERO,
    /* Receive (interleaved) data from Data from Port A and B using separate
       receive (Rx) channels with no extended zeros (14-bits) */
    ADI_BF506FADC1_DATA_DOUT_A_B_SEPERATE_RX,
    /* Receive (interleaved) data from Data from Port A and B using separate
       receive (Rx) channels with extended zeros (16-bits) */
    ADI_BF506FADC1_DATA_DOUT_A_B_SEPERATE_RX_XTND_ZERO,
    /* Receive (interleaved) data from Data from Port A and B using
       a single receive (Rx) channel with extended zeros (16-bits) */
    ADI_BF506FADC1_DATA_DOUT_A_B_SHARED_RX

} ADI_BF506FADC1_DATA_MODE;
```

5.7.2. Enumerations of ADC operating mode

```
typedef enum __AdcBf506fadc1OpMode
{
    /* Operate ADC in Normal mode */
    ADI_BF506ADC1_MODE_NORMAL = 0,
    /* Operate ADC in Partial Power-down mode */
    ADI_BF506ADC1_MODE_PARTIAL_POWERDOWN,
    /* Operate ADC in Complete Power-down mode */
    ADI_BF506ADC1_MODE_POWERDOWN

} ADI_BF506FADC1_OP_MODE;
```

5.8. Data Structures

5.8.1. Structure to pass GPIO flag IDs as ADC control signals

```
typedef struct __AdiBf506fadc1CtrlFlagPins
{
    /* ADC Channel select (A0) Flag */
    ADI_FLAG_ID     eA0Flag;

    /* ADC Channel select (A1) Flag */
    ADI_FLAG_ID     eA1Flag;

    /* ADC Channel select (A2) Flag */
    ADI_FLAG_ID     eA2Flag;

    /* ADC Logic select Flag */
    ADI_FLAG_ID     eRangeSelFlag;

    /* ADC Logic select Flag */
    ADI_FLAG_ID     eLogicSelFlag;

} ADI_BF506FADC1_CTRL_FLAG_PINS;
```

5.8.2. Structure to pass value/state of GPIO ADC controls

```
/*
** Structure to pass value/state of ADC control signals when the driver is configured to use
** GPIO Flag IDs to control ADC
*/
typedef struct __AdiBf506fadc1CtrlFlagState
{
    /* ADC Channel select (A0) Flag State
       'true' to set, 'false' to clear */
    bool    bSetA0;

    /* ADC Channel select (A1) Flag State
       'true' to set, 'false' to clear */
    bool    bSetA1;

    /* ADC Channel select (A2) Flag State
       'true' to set, 'false' to clear */
    bool    bSetA2;

    /* ADC Logic select Flag State
       'true' to set, 'false' to clear */
    bool    eSetRangeSel;

    /* ADC Logic select Flag State
       'true' to set, 'false' to clear */
    bool    eSetLogicSel;

} ADI_BF506FADC1_CTRL_FLAG_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.

- **`ADI_BF506ADC1_EntryPoint`**

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 Number	0	0 or 1	**ADI_BF506ADC1_CMD_SET_SPORT_DEV_NUMBER**
ACM Device Number	0	0	**ADI_BF506ADC1_CMD_SET_ACM_DEV_NUMBER**
GPIO flag id to control ADC	**ADI_FLAG_UNDEFINED**	Values of type **ADI_FLAG_ID**	**ADI_BF506ADC1_CMD_SET_ADC_CTRL_FLAG_PINS**
ADC Operating Mode	**ADI_BF506ADC1_MODE_POWERDOWN**	Enumeration of type **ADI_BF506ADC1_OP_MODE**	**ADI_BF506ADC1_CMD_SET_OPERATING_MODE**
ADC Serial Data Mode	**ADI_BF506FADC1_DATA_DOUT_A_ONLY**	Enumeration of type **ADI_BF506FADC1_DATA_MODE**	**ADI_BF506ADC1_CMD_SET_SERIAL_DATA_MODE**
ADC sampling rate / ACLK frequency	**20000000**	Depends on control device	**ADI_BF506ADC1_CMD_SET_SAMPLE_RATE_ACLK_FREQ**

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
Open ADC Cotrol Device	true/false ('true' to open, 'false' to close)	**ADI_BF506ADC1_CMD_OPEN_ADC_CTRL_DEV**
Open/Close SPORT	true/false ('true' to open, 'false' to close)	**ADI_BF506ADC1_CMD_OPEN_SPORT_DEV**

Table 5 – Additional Required Settings

7. Hardware Considerations

If BF506F ADC driver is set to use GPIO flag pins to control ADC signals, then application has to pass in the Flag pin ids and corresponding pin state before enabling ADC dataflow.

8. Appendix

8.1. Using BF506F ADC Device Driver in Applications

This section explains how to use BF506F ADC device driver in an application.

8.1.1. Interrupt Manager Data memory allocation

This section explains Interrupt manager memory allocation requirements for applications using this driver. Application must allocate memory for tow secondary interrupts, of size **ADI_INT_SECONDARY_MEMORY,** to handle SPORT Rx DMA channel. Additional memory of size **ADI_INT_SECONDARY_MEMORY** must be provided when the client decides to enable SPORT error reporting.

8.1.2. DMA Manager Data memory allocation

This section explains DMA manager memory allocation requirements for applications using this driver. The application should allocate base memory + memory for on SPORT DMA channel + memory for DMA channels used by other devices in the application

8.1.3. 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 SPORT device + memory for one BF506F ADC device + memory for other devices used by the application

8.1.4. Typical usage of BF506F ADC driver with ACM

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

a. Initialising BF506F ADC driver

Step 1: Open BF506F ADC Device driver with device specific entry point (refer section 6.1 for valid entry point)

Step 2: Set SPORT device number to be used to receive data from BF506F ADC

```
/* Example: Set SPORT Device number */
adi_dev_Control (hBf506fadc1Driver, ADI_BF506ADC1_CMD_SET_SPORT_DEV_NUMBER, (void *) 0);
```

Step 3: Open ACM (ADC Control device)

```
/* Example: Open ACM (ADC Control device) */
adi_dev_Control (hBf506fadc1Driver, ADI_BF506ADC1_CMD_OPEN_ADC_CTRL_DEV, (void *) true);
```

Step 4: Open SPORT Device

```
/* Example: Open ACM (ADC Control device) */
adi_dev_Control (hBf506fadc1Driver, ADI_BF506ADC1_CMD_OPEN_SPORT_DEV, (void *) true);
```

Step 5: Set SPORT Pin Mux settings (if required)

```
/* Example: Use SPORT DR1SEC on PG8 */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_SPORT_CMD_SET_PIN_MUX_MODE,
                 (void *) ADI_SPORT_PIN_MUX_MODE_1);
```

b. Configure ADC/ACM settings

Step 6: Set ACM ACLK frequency

```
/* Example: Set ACM ACLK frequency to 20MHz */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_BF506ADC1_CMD_SET_SAMPLE_RATE_ACLK_FREQ,
                 (void *) 20000000);
```

Step 7: Set ADC serial data mode

```
/* Example: Set ADC serial data mode to receive Data on DOUTA only */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_BF506ADC1_CMD_SET_SERIAL_DATA_MODE,
                 (void *) ADI_BF506FADC1_DATA_DOUT_A_ONLY);
```

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

```
/* Example: Set dataflow method */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_DEV_CMD_SET_DATAFLOW_METHOD,
                 (void *) ADI_DEV_MODE_CHAINED);
```

Step 9: Set ADC operating mode

a. Submit a dummy data buffer so that SPORT device can generate clock/frame sync to apply the operating mode. Without a buffer, SPORT will get into an infinite loop since the DMA manager wouldn't have a valid buffer to store data from sport.

```
/* Example: Submit a dummy buffer to power-up ADC */
adi_dev_Read (hBf506fadc1Driver,
              ADI_DEV_1D,
              (ADI_DEV_BUFFER *) &oAdcBuffer);
```

b. Set ADC operating mode to normal

```
/* Example: Set ADC in Normal Mode */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_BF506ADC1_CMD_SET_OPERATING_MODE,
                 (void *) ADI_BF506ADC1_MODE_NORMAL);
```

Step 11: Configure ACM registers using ACM service specific commands. Application can issue ACM service specific commands using the BF506F ADC driver handle

```
/******* Example ********/

/* Configure ACM in interrupt mode by providing a valid callback function */
adi_dev_control (hBf506fadc1Driver,
                 ADI_ACM_CMD_SET_CALLBACK_FN,
                 (void *) AcmCallback);


/* Configure ACM Timer */
adi_dev_control (hBf506fadc1Driver,
                 ADI_ACM_CMD_CONFIG_TMR,
                 (void *) &oAcmTmrConfig);

/* Configure ACM Event parameters */
adi_dev_control (hBf506fadc1Driver,
                 ADI_ACM_CMD_SET_EVENT_PARAMS_TABLE,
                 (void *) & oEventParamTable);
```

c: Submit buffers and enable dataflow

Step 12: Submit data buffers to store BF506F ADC samples
```
/* Example: Submit Inbound data Buffer */
adi_dev_Read(hBf506fadc1Driver, ADI_DEV_CIRC, (ADI_DEV_BUFFER *)  &oAdcBuffer);
```

Step 13: Enable BF506F ADC dataflow
```
/* Example:  Enable ADC Dataflow */
adi_dev_Control(hBf506fadc1Driver, ADI_DEV_CMD_SET_DATAFLOW, (void *)true);
```

Step 14: Issue/wait for an ACM trigger
```
/* Example:  Enable GP Timer to generate ACM Trigger */
adi_tmr_GPControl (ADI_TMR_GP_TIMER_7, ADI_TMR_GP_CMD_ENABLE_TIMER, (void *)true);
```

Step15: Service buffer processed callback within ADC driver callback routine, and ACM event callbacks in ACM service callback routine

d: Close BF506F ADC driver

Step16: Disable ADC dataflow
```
/* Example: Disable ADC dataflow */
adi_dev_Control(hBf506fadc1Driver, ADI_DEV_CMD_SET_DATAFLOW, (void *)false);
```

Step 17: Close ADC Driver
```
/* Example: Close ADC driver */
adi_dev_Close (hBf506fadc1Driver);
```

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

8.1.5. Typical usage of BF506F ADC driver with GPIO Flag control

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

a. Initialising BF506F ADC driver

Step 1: Open BF506F ADC Device driver with device specific entry point (refer section 6.1 for valid entry point)

Step 2: Set SPORT device number to be used to receive data from BF506F ADC
```
/* Example: Set SPORT Device number */
adi_dev_Control (hBf506fadc1Driver, ADI_BF506ADC1_CMD_SET_SPORT_DEV_NUMBER, (void *) 0);
```

Step 3: Set Flag IDs of ADC control signals and configure the driver to use GPIO flags to control ADC

```
/********** Example ***********/

/* Instance to hold Flag pins connected to ADC Control */
ADI_BF506FADC1_CTRL_FLAG_PINS   oFlagPins;

/* A0 control flag */
oFlagPins.eA0Flag          = ADI_FLAG_PG1;
/* A1 control flag */
oFlagPins.eA1Flag          = ADI_FLAG_PG2;
/* No control flag for A2 – assume the value is fixed in hardware */
oFlagPins.eA2Flag          = ADI_FLAG_UNDEFINED;
/* No Range select flag – assume the value is fixed in hardware */
oFlagPins.eRangeSelFlag    = ADI_FLAG_UNDEFINED;
/* Logic select flag
oFlagPins.eLogicSelFlag    = ADI_FLAG_PG3;
```

```
/* Set ADC control flag IDs */
adi_dev_control (hBf506fadc1Driver,
                 ADI_BF506ADC1_CMD_SET_ADC_CTRL_FLAG_PINS,
                 (void *) &oFlagPins);
```

Step 4: Open ADC Control device (GPIO flag pins)

```
/* Example: Open ADC Control device */
adi_dev_Control (hBf506fadc1Driver, ADI_BF506ADC1_CMD_OPEN_ADC_CTRL_DEV, (void *) true);
```

Step 5: Open SPORT Device

```
/* Example: Open ACM (ADC Control device) */
adi_dev_Control (hBf506fadc1Driver, ADI_BF506ADC1_CMD_OPEN_SPORT_DEV, (void *) true);
```

Step 6: Set SPORT Pin Mux settings (if required)

```
/* Example: Use SPORT DR1SEC on PG8 */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_SPORT_CMD_SET_PIN_MUX_MODE,
                 (void *) ADI_SPORT_PIN_MUX_MODE_1);
```

b. Configure ADC/GPIO control settings

Step 7: Set ADC sampling rate frequency

```
/* Example: Set ADC sample rate as 100KHz */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_BF506ADC1_CMD_SET_SAMPLE_RATE_ACLK_FREQ,
                 (void *) 100000);
```

Step 8: Set ADC serial data mode

```
/* Example: Set ADC serial data mode to receive Data on DOUTA only */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_BF506ADC1_CMD_SET_SERIAL_DATA_MODE,
                 (void *) ADI_BF506FADC1_DATA_DOUT_A_ONLY);
```

Step 9: Set dataflow method (this will be passed to SPORT driver)

```
/* Example: Set dataflow method */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_DEV_CMD_SET_DATAFLOW_METHOD,
                 (void *) ADI_DEV_MODE_CHAINED);
```

Step 10: Set ADC operating mode

a. Submit a dummy data buffer so that SPORT device can generate clock/frame sync to apply the operating mode. Without a buffer, SPORT will get into an infinite loop since the DMA manager wouldn't have a valid buffer to store data from sport.

```
/* Example: Submit a dummy buffer to power-up ADC */
 adi_dev_Read (hBf506fadc1Driver,
                         ADI_DEV_1D,
                         (ADI_DEV_BUFFER *) &oAdcBuffer);
```

b. Set ADC operating mode to normal

```
/* Example: Set ADC in Normal Mode */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_BF506ADC1_CMD_SET_OPERATING_MODE,
                 (void *) ADI_BF506ADC1_MODE_NORMAL);
```

Step 11: Set value/state of GPIO flags used to control ADC

```
/********** Example ***********/

/* Instance to hold value of ADC control flag pins */
 ADI_BF506FADC1_CTRL_FLAG_STATE   oFlagState;
```

```
/* ADC control pin state */
oFlagState.bSetA0   = true;
oFlagState.bSetA1   = false;
oFlagState.bSetA2   = false;
oFlagState.eSetRangeSel    = false;
oFlagState.eSetLogicSel    = true;

/* Pass GPIO flag state to ADC driver */
adi_dev_Control (hBf506fadc1Driver,
                 ADI_BF506ADC1_CMD_SET_ADC_CTRL_FLAG_STATE,
                 (void *) &oFlagState);
```

c: Submit buffers and enable dataflow

Step 12: Submit data buffers to store BF506F ADC samples

```
/* Example: Submit Inbound data Buffer */
adi_dev_Read(hBf506fadc1Driver, ADI_DEV_CIRC, (ADI_DEV_BUFFER *)  &oAdcBuffer);
```

Step 13: Enable BF506F ADC dataflow

```
/* Example:  Enable ADC Dataflow */
adi_dev_Control(hBf506fadc1Driver, ADI_DEV_CMD_SET_DATAFLOW, (void *)true);
```

Step14: Service buffer processed callback within ADC driver callback routine

d: Close BF506F ADC driver

Step15: Disable ADC dataflow

```
/* Example: Disable ADC dataflow */
adi_dev_Control(hBf506fadc1Driver, ADI_DEV_CMD_SET_DATAFLOW, (void *)false);
```

Step 16: Close ADC Driver

```
/* Example: Close ADC driver */
adi_dev_Close (hBf506fadc1Driver);
```

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