ANALOG DEVICES

# ADI_TWI DEVICE DRIVER

DATE: 3 JANUARY 2011

## Table of Contents

## Table of Figures

**Document Revision History**

| Date | Description of Changes |
|---|---|
| 1/3/2011 | Documented support for Repeated Start Condition |
| 7/2/2006 | Updated document format |
| 5/18/06 | Updated file locations |

**Table 1 - Revision History**

## 1. Overview

The TWI device driver can operate in hardware or pseudo modes, this document only describes the hardware operation of the TWI driver. For information on the pseudo TWI please refer to the ADI_TWI_PSEUDO Device Driver Documentation manual. The TWI device driver supports both master and slave modes of operation.

# 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/twi/adi_twi.h> This file contains all definition, function prototypes etc. for the TWI driver itself.

## 2.2. Source Files

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

- <Blackfin/lib/src/drivers/twi/adi_twi.c> This file contains all source code for the TWI driver.

## 3. Lower Level Drivers

This driver does not use any lower level driver.

# 4. Resources Required

Device drivers typically consume some amount of system resources. This section describes the resources required by the device driver.

Unless explicitly noted in the sections below, this device driver uses the System Services to access and control any required hardware. The information in this section may be helpful in determining the resources this driver requires, such as the number of interrupt handlers or number of DMA channels etc., from the System Services.

Because dynamic memory allocations are not used in the Device Drivers or System Services, all memory used by the Device Drivers and System Services must be supplied by the application. The Device Drivers and System Services supply macros that can be used by the application to size the amount of base memory and/or the amount of incremental memory required to support the needed functionality. Memory for the Device Manager and System Services is provided in the initialization functions (adi_xxx_Init()).

## 4.1. Interrupts

This driver uses one interrupt, the TWI hardware interrupt.

## 4.2. DMA

This driver does not use DMA.

## 4.3. Timers

This driver does not use any timers.

## 4.4. Real-Time Clock

This driver does not use the real-time clock.

## 4.5. Programmable Flags

This driver does not use and programmable flags.

## 4.6. Pins

This driver uses the SCL and SDA pins. These vary depending to the Blackfin device used, please check hardware reference manual for the relevant pins used.

Note: not all blackfin devices have a built-in hardware TWI. If your device does not have one you can use the TWI Pseudo driver to emulate the device onto the general purpose pins. Check 'TWIPseudoDeviceDriverDocumentation' for details on the Pseudo TWI.

# 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_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_CHAINED | Supports the chained buffer method. This puts the TWI device into slave mode. |
| ADI_DEV_MODE_SEQ_CHAINED | Supports the sequential I/O dataflow method. This puts the TWI device into master mode. |

**Table 3 - Supported Dataflow Methods**

## 5.3. Buffer Types

The driver supports the buffer types listed in the table below.

***Master-Mode Repeated Start Condition***
*During TWI transfers there is often a need to first send a command and then read back an answer right away. This has to be done without the risk of another (multi-master) device interrupting this atomic operation. The TWI protocol defines a so-called "repeated start condition" to accomplish this. After having sent the address byte (address and read/write bit) the master may send any number of bytes followed by a stop condition. Instead of sending the stop condition it is also allowed to send another (repeated) start condition, again followed by an address (and of course including a read/write bit) and more data. This is defined recursively allowing any number of (repeated) start conditions to be sent. The purpose of this is to allow combined write/read operations to one or more devices without releasing the bus and thus guaranteeing that the operation is not interrupted. Regardless of the number of start conditions sent during one transfer, the transfer must be ended by exactly one stop condition.*

*In the case of the 1D sequential buffer type (ADI_DEV_SEQ_1D_BUFFER), users may configure a repeated start condition by bitwise ORing in the predefined ADI_TWI_REPEAT_START macro with the required slave device address and setting the result as the buffer's pAdditionalInfo value. The last such buffer must not use the repeated start condition so as to insure the sequence ends with the required stop condition.*

- ADI_DEV_1D_BUFFER
  - Linear one-dimensional buffer
  - pAdditionalInfo – ignored
- ADI_DEV_SEQ_1D_BUFFER
  - Linear one-dimensional sequential buffer
  - pAdditionalInfo – u32 (used to pass the master-mode slave device address value in the range 8 to 123, optionally ORed with the ADI_TWI_REPEAT_START macro *if and only if requesting use of the repeated start condition during the transfer*)

## 5.4. Command IDs

This section enumerates the commands that are supported by the driver. The commands are divided into three sections. The first section describes commands that are supported directly by the Device Manager. The next section describes common commands that the driver supports. The remaining section describes driver specific commands.

Commands are sent to the device driver via the adi_dev_Control() function. The adi_dev_Control() function accepts three arguments:

- DeviceHandle – This parameter is a ADI_DEV_DEVICE_HANDLE type that uniquely identifies the device driver. This handle is provided to the client in the adi_dev_Open() function call.
- CommandID – This parameter is a u32 data type that specifies the command ID.
- Value – This parameter is a void * whose value is context sensitive to the specific command ID.

The sections below enumerate the command IDs that are supported by the driver and the meaning of the Value parameter for each command ID.

### 5.4.1. Device Manager Commands

The commands listed below are supported and processed directly by the Device Manager. As such, all device drivers support these commands.

- ADI_DEV_CMD_TABLE
    - Table of command pairs being passed to the driver
    - Value – ADI_DEV_CMD_VALUE_PAIR *
- ADI_DEV_CMD_END
    - Signifies the end of a command pair table
    - Value – ignored
- ADI_DEV_CMD_PAIR
    - Single command pair being passed
    - Value – ADI_DEV_CMD_PAIR *
- ADI_DEV_CMD_SET_SYNCHRONOUS
    - Enables/disables synchronous mode for the driver
    - Value – TRUE/FALSE

### 5.4.2. Common Commands

The command IDs described in this section are common to many device drivers. The list below enumerates all common command IDs that are supported by this device driver.

- ADI_DEV_CMD_SET_DATAFLOW_METHOD
    - Specifies the dataflow method the device is to use. The list of dataflow types supported by the device driver is specified in section 5.2.
    - Value – ADI_DEV_MODE enumeration
- ADI_DEV_CMD_SET_DATAFLOW
    - Enables/disables dataflow through the device
    - Value – TRUE/FALSE
- ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT
    - Determines if the device driver is supported by peripheral DMA
    - Value – u32 * (location where TRUE or FALSE is stored)

### 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_TWI_CMD_SET_HARDWARE
    - Sets the TWI hardware the driver uses, this is limited to the hardware availiable
    - Value – ADI_INT_PERIPHERAL_ID (Interrupt id of TWI hardware)
- ADI_TWI_CMD_SET_PSEUDO
    - Sets the TWI in pseudo mode, please refer to TWIPseudoDeviceDriverDocumentation for more device operation in this mode.
    - Value – struct adi_twi_pseudo_port * (Pseudo port structure, defined in adi_twi.h)
- ADI_TWI_CMD_SET_RATE
    - Sets the frequency of the TWI port
    - Value – struct adi_twi_bit_rate * (structure containig  frequency and duty cycle, defined in adi_twi_h)
- ADI_TWI_CMD_SET_SLAVE_ADDR
    - Sets the address used when the device driver is operating in slave mode
    - Value – u32 (Address, this is a 7bit value)
- ADI_TWI_CMD_SET_GCAL
    - Enables general call addressing in slave mode
    - Value – not used

- ADI_TWI_CMD_SET_SCCB
    - Enables SCCB compatibility in master mode.
    - Value – not used
- ADI_TWI_CMD_SET_SCLOVR
    - Overrides the SCL flag, when set the output on the SCL pin will be driven low
    - Value – TRUE/FALSE
- ADI_TWI_CMD_SET_SDAOVR
    - Overrides the SDA flag, when set the output on the SDA pin will be driver low
    - Value – TRUE/FALSE
- ADI_TWI_CMD_SET_FIFO
    - Sets the FIFO buffer operating mode
    - Value – u32 (0x0008 enables receive interrupts on word boundary, 0x0004 enables word transfers on transmit interrupts, the driver automatically handles odd byte transfers)
- ADI_TWI_CMD_SET_LOSTARB
    - Set the number of retries for lost arbitration events before callback is run
    - Value – u32 (0=never generate callback, otherwise value indicates the number of events needed to trigger callback)
- ADI_TWI_CMD_SET_ANAK
    - Set the number of retries for address negative acknowlege events before callback is run
    - Value – u32 (0=never generate callback, otherwise value indicates the number of events needed to trigger callback)
- ADI_TWI_CMD_SET_DNAK
    - Set the number of retries for data negative acknowledge  events before callback is run
    - Value – u32 (0=never generate callback, otherwise value indicates the number of events needed to trigger callback)
- ADI_TWI_CMD_GET_SENSE
    - Get the status of SDA, SCL and BUSBUSY flags
    - Value – u32 * (Address of value holding status bits, 0x0100=BUSBUSY, 0x0080=SCL and 0x0040=SDA)

## 5.5. Callback Events

This section enumerates the callback events the device driver is capable of generating.  The events are divided into two sections.  The first section describes events that are common to many device drivers.  The next section describes driver specific event IDs.  The callback function of the client should be prepared to process each of the events in these sections.

The callback function is of the type ADI_DCB_CALLBACK_FN.  The callback function is passed three parameters.  These parameters are:

- ClientHandle – This void * parameter is the value that is passed to the device driver as a parameter in the adi_dev_Open() function.
- EventID – This is a u32 data type that specifies the event ID.
- Value – This parameter is a void * whose value is context sensitive to the specific event ID.

The sections below enumerate the event IDs that the device driver can generate and the meaning of the Value parameter for each event ID.

### 5.5.1. Common Events

The events described in this section are common to many device drivers. The list below enumerates all common event IDs that are supported by this device driver.

- ADI_DEV_EVENT_BUFFER_PROCESSED
    - Notifies callback function that a chained or sequential I/O buffer has been processed by the device driver. This event is also used to notify that an entire circular buffer has been processed if the driver was directed to generate a callback upon completion of an entire circular buffer.
    - Value – For chained or sequential I/O dataflow methods, this value is the CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_Read(), adi_dev_Write() or adi_dev_SequentialIO() function. For the circular dataflow method, this value is the address of the buffer provided in the adi_dev_Read() or adi_dev_Write() function.

### 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.

Master mode events
- ADI_TWI_EVENT_BUFWRERR
    - Receive buffer overflow, usually indicates no read buffer availiable
    - Value - CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_SequentialIO() function.
- ADI_TWI_EVENT_BUFRDERR
    - Transmit buffer underflow, usually indicates to write buffer avvailiable
    - Value - CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_SequentialIO() function.
- ADI_TWI_EVENT_DNAK
    - Data Negative Acknowledge has been received x times (where x is value set by ADI_TWI_CMD_SET_DNAK)
    - Value - CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_SequentialIO() function.
- ADI_TWI_EVENT_ANAK
    - Address Negative Acknowledge has been received x times (where x is value set by ADI_TWI_CMD_SET_ANAK)
    - Value - CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_SequentialIO() function.
- ADI_TWI_EVENT_LOSTARB
    - TWI has Lost Bus Arbitration x times (where x is value set by ADI_TWI_CMD_SET_LOSTARB)
    - Value - CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_SequentialIO() function.
- ADI_TWI_EVENT_YANKED
    - Buffer has been removed from device transmit or receive queue, this occurs if there is an error sending/receiving a buffer and there are subsequent buffers that are chaing with repeat start address
    - Value - Buffer that has been removed

Slave mode events
- ADI_TWI_EVENT_XMT_NEED
    - Driver has transmitted all data and needs a new buffer
    - Value – Not used
- ADI_TWI_EVENT_XMT_PROCESSED
    - Buffer passed to driver with adi_dev_Write() function has been processed.
    - Value - CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_Write() function.
- ADI_TWI_EVENT_RCV_NEED

  - Driver has received a byte in slave mode and needs a buffer to put data into
  - Value - Not used
- ADI_TWI_EVENT_RCV_PROCESSED
  - Buffer passed to driver with adi_dev_Read() function has been processed
  - Value - CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_Read() function.
- ADI_TWI_EVENT_SERR
  - The read buffer has been returned with an unexpected stop.
  - Value – Not used
- ADI_TWI_EVENT_SOVF
  - The read buffer has data from two transfers
  - Value – Not used
- ADI_TWI_EVENT_GCALL
  - Data has been received from a General Call address
  - Value - CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_Read(), adi_dev_Write() or adi_dev_SequentialIO() function.

## 5.6. Return Codes

All API functions of the device driver return status indicating either successful completion of the function or an indication that an error has occurred. This section enumerates the return codes that the device driver is capable of returning to the client. A return value of ADI_DEV_RESULT_SUCCESS indicates success, while any other value indicates an error or some other informative result. The value ADI_DEV_RESULT_SUCCESS is always equal to the value zero. All other return codes are a non-zero value.

The return codes are divided into two sections. The first section describes return codes that are common to many device drivers. The next section describes driver specific return codes. Wherever functions in the device driver API are called, the application should be prepared to process any of these return codes.

Typically the application should check the return code for ADI_DEV_RESULT_SUCCESS, taking appropriate corrective action if ADI_DEV_RESULT_SUCCESS is not returned. For example:

```
if (adi_dev_Xxxx(…) == ADI_DEV_RESULT_SUCCESS) {
      // normal processing
} else {
      // error processing
}
```

### 5.6.1. Common Return Codes

The return codes described in this section are common to many device drivers. The list below enumerates all common return codes that are supported by this device driver.

- ADI_DEV_RESULT_SUCCESS
  - The function executed successfully.
- ADI_DEV_RESULT_NOT_SUPPORTED
  - The function is not supported by the driver.
- ADI_DEV_RESULT_DEVICE_IN_USE
  - The requested device is already in use.
- ADI_DEV_RESULT_NO_MEMORY
  - There is insufficient memory available.
- ADI_DEV_RESULT_BAD_DEVICE_NUMBER
  - The device number is invalid.
- ADI_DEV_RESULT_DIRECTION_NOT_SUPPORTED
  - The device cannot be opened in the direction specified.
- ADI_DEV_RESULT_BAD_DEVICE_HANDLE

  - The handle to the device driver is invalid.
- ADI_DEV_RESULT_BAD_MANAGER_HANDLE
  - The handle to the Device Manager is invalid.
- ADI_DEV_RESULT_BAD_PDD_HANDLE
  - The handle to the physical driver is invalid.
- ADI_DEV_RESULT_INVALID_SEQUENCE
  - The action requested is not within a valid sequence.
- ADI_DEV_RESULT_ATTEMPTED_READ_ON_OUTBOUND_DEVICE
  - The client attempted to provide an inbound buffer for a device opened for outbound traffic only.
- ADI_DEV_RESULT_ATTEMPTED_WRITE_ON_INBOUND_DEVICE
  - The client attempted to provide an outbound buffer for a device opened for inbound traffic only.
- ADI_DEV_RESULT_DATAFLOW_UNDEFINED
  - The dataflow method has not yet been declared.
- ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE
  - The dataflow method is incompatible with the action requested.
- ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE
  - The device does not support the buffer type provided.
- ADI_DEV_RESULT_CANT_HOOK_INTERRUPT
  - The Interrupt Manager failed to hook an interrupt handler.
- ADI_DEV_RESULT_CANT_UNHOOK_INTERRUPT
  - The Interrupt Manager failed to unhook an interrupt handler.
- ADI_DEV_RESULT_NON_TERMINATED_LIST
  - The chain of buffers provided is not NULL terminated.
- ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED
  - No callback function was supplied when it was required.
- ADI_DEV_RESULT_REQUIRES_UNIDIRECTIONAL_DEVICE
  - Requires the device be opened for either inbound or outbound traffic only.
- ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE
  - Requires the device be opened for bidirectional traffic only.

### 5.6.2. Device Driver Specific Return Codes

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

- ADI_TWI_RESULT_BAD_RATE
  - The data rate passed using ADI_TWI_CMD_SET_RATE is outside valid values, the frequency must be within 20 to 400 and duty cycle must be within 1 to 99. Depending on the clock speed of the Blackfin other values might generate this error.
- ADI_TWI_RESULT_BAD_ADDR
  - The address pass in either slave mode (with ADI_TWI_CMD_SET_SLAVE) or in master mode (in pAdditionalInfo) is invalid. The address must be within the range 8 to 123 (optionally ORed with the ADI_TWI_RSTART_ADDR macro if using the *repeated start condition* – see description in the Buffer Types section).
- ADI_TWI_RESULT_NO_DATA
  - The read/write/sequential buffer submitted contained no data.
- ADI_TWI_RESULT_SLAVE_DCB
  - The device has been configured for slave mode and uses deferred callbacks. This could cause problems because whilst the driver is waiting for another buffer to transmit/receive data from/too the bus is being held by the twi driver.
- ADI_TWI_RESULT_BAD_PSEUDO
  - The pseudo port configuration cannot be setup, because the values supplied are invalid or the timers and/or flags are already being used.

# 6. Opening and Configuring the Device Driver

This section describes the default configuration settings for the device driver and any additional configuration settings required from the client application.

## 6.1. Entry Point

When opening the device driver with the adi_dev_Open() function call, the client passes a parameter to the function that identifies the specific device driver that is being opened. This parameter is called the entry point. The entry point for this driver is listed below.

- <ADITWIEntryPoint>

## 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. Any configuration settings not listed in the table below are undefined.

| Item | Default Value | Possible Values | Command ID |
|---|---|---|---|
| Hardware configuration | ADI_INT_TWI | Depends on Blakfin device used | ADI_TWI_CMD_SET_HARDWARE |
| Rate | Frequency=100<br>Duty cycle=50 | Frequency 20-400<br>Duty cycle 1-99 | ADI_TWI_CMD_SET_RATE |
| FIFO | 0x000C | 0x0000, 0x0004, 0x0008 and 0x000C | ADI_TWI_CMD_SET_FIFO |
| Lost arbitration | 3 | 0 - 255 | ADI_TWI_CMD_SET_LOSTARB |
| ANAK | 1 | 0 – 255 | ADI_TWI_CMD_SET_ANAK |
| DNAK | 1 | 0 – 255 | ADI_TWI_CMD_SET_DNAK |
| SCL override | 0 | 0 - 1 | ADI_TWI_CMD_SET_SCLOVR |
| SDA override | 0 | 0 – 1 | ADI_TWI_CMD_SET_SDAOVR |
| General call | 0 | 1 (Once set cannot be reset) | ADI_TWI_CMD_SET_GCALL |
| SCCB mode | 0 | 1 (Once set cannot be reset) | ADI_TWI_CMD_SET_SCCB |

**Table 4 - Default Settings**

## 6.3. Additional Required Configuration Settings

In addition to the possible overrides of the default driver settings, the device driver requires the application to specify the additional configuration information listed in the table below.

| Item | Possible Values | Command ID |
|---|---|---|
| Dataflow method | See section 5.2 | ADI_DEV_CMD_SET_DATAFLOW_METHOD |

**Table 5 – Additional Required Settings**

# 7. Hardware Considerations

The TWI device driver does not require any hardware configuration, however to use the hardware TWI device the Blackfin device used must support hardware TWI (ie BF537). Listed below are the twi data structures that are used to configure the pseudo port and the twi data rate along with some typlical configuration options for the TWI.

Note: the SCL and SDA pins need an external pull-up resistor, please refer to the TWI specifications for more information about this, the ADSP-BF537 EZ-Kit Lite has these resistors on the board.

## 7.1. TWI Pseudo Port Structure

Parameter passed with ADI_TWI_CMD_SET_PSEUDO.

```
// stucture for software based TWI port
typedef struct
{
        ADI_FLAG_ID scl;                        // Flag that is to be used as SCL port
        ADI_FLAG_ID sda;                        // Flag that is to be used as SDA port
        u32 timer;                              // Timer that is used to clock the pseudo TWI
        ADI_INT_PERIPHERAL_ID scl_pid;          // Clock interrupt id
} adi_twi_pseudo_port;                          // submit as ADI_TWI_CMD_SET_PSEUDO control data
```

## 7.2. TWI bit rate structure

Parameter passed with ADI_TWI_CMD_SET_RATE.

```
// structure for TWI timing info
typedef struct
{
        u16  frequency;                         // The TWI bit rate per second in kHz
        u16  duty_cycle;                        // The clock duty cycle (not used in pseudo mode)
} adi_twi_bit_rate;                             // submit as ADI_TWI_CMD_SET_RATE control data
```

## 7.3. Hardware TWI master configuration

Configure the TWI as hardware master, with the bitrate set to 100kHz and 50% duty cycle, interrupts on bytes and drop buffer if LOSTARB condition is met.

```
        adi_twi_bit_rate rate={100,50};
        ADI_DEV_CMD_VALUE_PAIR config[]={               // configuration table for the TWI driver
                {ADI_TWI_CMD_SET_HARDWARE,(void *)ADI_INT_TWI},
                {ADI_DEV_CMD_SET_DATAFLOW_METHOD,(void *)ADI_DEV_MODE_SEQ_CHAINED},
                {ADI_TWI_CMD_SET_FIFO,(void *)0x0000},
                {ADI_TWI_CMD_SET_RATE,(void *)(&rate)},
                {ADI_TWI_CMD_SET_LOSTARB,(void *)1},
                {ADI_TWI_CMD_SET_ANAK,(void *)0},
                {ADI_TWI_CMD_SET_DNAK,(void *)0},
                {ADI_DEV_CMD_SET_DATAFLOW,(void *)TRUE},
                {ADI_DEV_CMD_END,NULL}
        };
        adi_dev_Open(
                ManagerHandle,
                &ADITWIEntryPoint,
                DeviceNumber,
                DeviceHandle,
                &PhysicalDevice,
                ADI_DEV_DIRECTION_BIDIRECTIONAL,
                NULL,
                DCBHandle,
                DMCallback);
        adi_dev_Control(device,ADI_DEV_CMD_TABLE,config);
```

## 7.4. Hardware TWI slave configuration

Configure the TWI as hardware slave. The TWI is setup to interrupt on words and the slave address is 0x60.

```
ADI_DEV_CMD_VALUE_PAIR config_slave[]={     // configuration table for the TWI driver
            {ADI_TWI_CMD_SET_HARDWARE,(void *)ADI_INT_TWI},
            {ADI_DEV_CMD_SET_DATAFLOW_METHOD,(void *)ADI_DEV_MODE_CHAINED},
            {ADI_TWI_CMD_SET_FIFO,(void *)0x000C},
            {ADI_TWI_CMD_SET_SLAVE_ADDR,(void *)0x60},
            {ADI_DEV_CMD_SET_DATAFLOW,(void *)TRUE},
            {ADI_DEV_CMD_END,NULL}
};
```