ANALOG DEVICES

ADI_AD7879 / ADI_AD7879_1
DEVICE DRIVER

DATE: DECEMBER 4, 2009

Table of Contents

List of Tables

Document Revision History

Date	Description of Changes
March 10, 2009	Initial release
December 1, 2009	Added description of support for AD7879-1.
December 4, 2009	Modified description of the header files.

Table 1 – Revision History

1. Overview

This driver provides an effective and easy way to manage Analog Devices' AD7879 / AD7879-1 Touch Screen controller. The AD7879 and AD7879-1 are identical except for the means of accessing their internal registers: the former uses an SPI connection while the latter uses TWI. In this document '*AD7879*' refers to either controller, except where differences between the controllers are explicitly described.

The driver can access AD7879 device registers or individual register fields via SPI (AD7879) or TWI (AD7879-1). The driver is also capable of monitoring AD7879 interrupt signals (PENIRQ, DAV & ALERT) and taking appropriate actions, which will reduce the application code overhead considerably.

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/touchscreen/adi_ad7879_common.h>
  - This file contains register and type definitions that are common to the AD7879 and AD7879-1. It should not be included directly from client code --- use adi_ad7879.h or adi_dev_ad7879_1.h as appropriate instead.

For the AD7879:

- <drivers/spi/adi_spi.h>
  - This file contains all definitions, function prototypes etc. specific to the SPI device
- <drivers/touchscreen/adi_ad7879.h>
  - This file contains includes the contents of adi_dev_ad7879_common.h and defines the AD7879 driver entry-point.

For the AD7879-1:

- <drivers/deviceaccess/adi_device_access.h>
  - This file contains definitions and prototypes for access the device via TWI.
- <drivers/touchscreen/adi_ad7879_1.h>
  - This file contains includes the contents of adi_dev_ad7879_common.h and defines the AD7879-1 driver entry-point.

2.2. Source Files

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

For the AD7879:

- < Blackfin/lib/src/drivers/touchscreen/adi_ad7879.c>
  - This file contains all the source code for the AD7879 device driver. All source code is written in ‘C’. There are no assembly level functions in this driver.

For the AD7879-1:

- < Blackfin/lib/src/drivers/touchscreen/adi_ad7879_1.c>
  - This file defines pre-processor identifier ADI_IMPLEMENT_AD7879_1 and then includes the main < Blackfin/lib/src/drivers/touchscreen/adi_ad7879.c> file.

3. 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, AD7879 device driver uses the System Services to perform the necessary low-level hardware access and control. This driver is built on top of Interrupt driven SPI driver. **Each AD7879 device requires two additional (driver) memory of size ADI_DEV_DEVICE_MEMORY, one for AD7879 and one for SPI or TWI (as appropriate).**

3.1. Interrupts

AD7879 is capable of generating one interrupt signal for each of the three modes available:. 1) Pen Alert 2) Data Available (DAV) and 3) Limit Comparison Alert (ALERT). These signals are usually mapped to a Blackfin GPIO port. It is the applications responsibility to provide additional, secondary, interrupt memory to accommodate this flag interrupt. **This driver requires additional memory for one flag interrupt and one SPI or TWI interrupt (as appropriate), where each interrupt requiring memory of size ADI_INT_SECONDARY_MEMORY. One additional memory of ADI_INT_SECONDARY_MEMORY size must be provided when client decides to enable SPI error reporting.**

3.2. DMA

AD7879 driver doesn't use or support DMA.

3.3. Timers

Timers are not used by this driver.

3.4. Real-Time Clock

Real-time clock is not used by this driver.

3.5. Pins

Connect Blackfin SPI port pins to AD7879 SPI port, or TWI port pins to AD7879-1 TWI port, as appropriate.

4. Supported Features of the Device Driver

This section describes what features are supported by the device driver.

4.1. Directionality of the SPI Driver

ADI_DEV_DIRECTION	Description
ADI_DEV_ DIRECTION_BIDIRECTIONAL	Supports both the reception of data and transmission of data through the device via SPI. This directionality is fixed and cannot be changed by the application.

Table 2 – Supported Dataflow Directions

4.2. Dataflow Method of the SPI Driver

ADI_DEV_MODE	Description
ADI_DEV_MODE_SEQ_CHAINED	Supports the sequential chained buffer method – **used for AD7879 register access via SPI or TWI.** This dataflow method is also fixed and cannot be changed by the application.

Table 3 – Supported Dataflow Methods

4.3. Buffer Types

AD7879 driver doesn't support buffer submission functions like adi_dev_Read(), adi_dev_Write and adi_dev_SequentialIO(). The application can use adi_dev_Control() function to access AD7879 device registers and register fields.

4.4. 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.

4.4.1. 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

4.4.2. 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.

Device Access Commands to access AD7879 registers

- ADI_DEV_CMD_REGISTER_READ
  - Reads a single device register (Refer to section 9.1.1 for example)
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics) (Refer section 8.1)
- ADI_DEV_CMD_REGISTER_FIELD_READ
  - Reads a specific field location in a single device register (Refer to section 9.1.2 for example)
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics) (Refer section 8.3)
- ADI_DEV_CMD_REGISTER_TABLE_READ
  - Reads a table of selective device registers (Refer to section 9.1.3 for example)
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics) (Refer section 8.1)
- ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ
  - Reads a table of selective device register fields (Refer to section 9.1.4 for example)
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics) (Refer section 8.3)
- ADI_DEV_CMD_REGISTER_BLOCK_READ
  - Reads a block of consecutive device registers (Refer to section 9.1.5 for example)
  - Value – ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics) (Refer section 8.2)
- ADI_DEV_CMD_REGISTER_WRITE
  - Writes to a single device register (Refer to section 9.2.1 for example)
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics) (Refer section 8.1)
- ADI_DEV_CMD_REGISTER_FIELD_WRITE
  - Writes to a specific field location in a single device register (Refer to section 9.2.2 for example)
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics) (Refer section 8.3)
- ADI_DEV_CMD_REGISTER_TABLE_WRITE
  - Writes to a table of selective device registers (Refer to section 9.2.3 for example)
  - Value – ADI_DEV_ACCESS_REGISTER * (register specifics) (Refer section 8.1)
- ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE
  - Writes to a table of selective device register fields (Refer to section 9.2.4 for example)
  - Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics) (Refer section 8.3)
- ADI_DEV_CMD_REGISTER_BLOCK_WRITE
  - Writes to a block of consecutive device registers (Refer to section 9.2.5 for example)
  - Value – ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics) (Refer section 8.2)

4.4.3. 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.

For AD7879:

- ADI_AD7879_CMD_SET_SPI_DEVICE_NUMBER
  - Sets Blackfin SPI Device Number to be used to access AD7879
  - Value – u8

- ADI_AD7879_CMD_SET_SPI_CS
    - Sets Blackfin SPI Slave select port used to select AD7879 device for SPI access
    - Value – u8

For AD7879-1:

- ADI_AD7879_CMD_SET_TWI_DEVICE_ADDRESSES
    - Sets TWI controller number and 7879-1 TWI device address to use to access AD7879 registers
    - Value – (u8 (TWI controller number) << 8) | (u8 (TWI device address))
- ADI_AD7879_CMD_SET_TWI_CONFIG_TABLE
    - Sets TWI Configuration table specific to the application
    - Value – pointer to ADI_DEV_CMD_VALUE_PAIR

For both AD7879 and AD7879-1:

- ADI_AD7879_CMD_ENABLE_INTERRUPT_PEN_INT_IRQ
    - Sets AD7879 driver to monitor PENIRQ interrupt signal
    - Value – ADI_AD7879_INTERRUPT_PORT*
- ADI_AD7879_CMD_DISABLE_INTERRUPT_PENIRQ
    - Remove PENIRQ interrupt from AD7879 driver interrupt monitoring list
    - Value – NULL

4.5. 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.

4.5.1. Common Events

There are no common event IDs supported by this driver

4.5.2. 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. Note that the controller has only one interrupt line. The controller can support three modes, however: 1) PEN Mode 2) Data Available Mode 3) Limit Comparison Alert (ALERT). The driver, depending on which mode the controller is in, creates mode specific callback events as follows.

- ADI_AD7879_EVENT_PENIRQ_TOUCH
    - Indicates that PENIRQ has detected a screen touch event.
    - Value – NULL

- ADI_AD7879_EVENT_PENIRQ_RELEASE
    - Indicates that PENIRQ has detected a screen release event.
    - Callback Argument - pointer to a structure of type ADI_AD7879_RESULT_REGS holding result value of selected register

ADI_AD7879_EVENT_SINGLE_DAV
/* 0x401A0008 - Callback Event occurs when AD7879
is configured in single channel mode and indicates
that a Data Available (DAV) interrupt has occurred.
Callback Argument - pointer to a location holding
result value of the selected channel. */

- Occurs when AD7879 is configured in single channel mode and indicates that a Data Available (DAV) interrupt has occurred
- Callback Argument - pointer to a location holding

- ADI_AD7879_EVENT_SEQUENCER_DAV
  - Occurs when AD7879 is configured in Master Sequencer mode and indicates that a Data Available(DAV) interrupt has occurred
  - Callback Argument - pointer to a structure of type ADI_AD7879_RESULT_REGS holding result value of

- ADI_AD7879_EVENT_ALARM_OFL
  - Occurs when AD7879 is configured in "Out of limits alarm interrupt mode" and indicates that a OFL interrupt has occurred.
  - This function is not supported by the EZkit, this Callback serves as indication only.

4.6. 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 */
}
```

4.6.1. 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

4.6.2. 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_AD7879_RESULT_CMD_NOT_SUPPORTED
    - Occurs when client issues a command which is not supported by this driver

5. 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.

5.1. 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.

For AD7879:

- ADIAD7879EntryPoint

For AD7879-1:

- ADIAD7879_1_EntryPoint

5.2. 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
AD7879 Control Register 1	0	Configuration dependent (0x000 to 0xFFF)	Use Device Access commands to configure Control Register 1
AD7879 Control Register 2	0	Configuration dependent (0x000 to 0xFFF)	Use Device Access commands to configure Control Register 2
AD7879 Alert Register	0	Configuration dependent (0x000 to 0xFFF)	Use Device Access commands to configure Alert Register
SPI Device Number	0	Processor dependent	ADI_AD7879_CMD_SET_SPI_DEVICE_NUMBER
SPI Chipselect	0	1 to 7	ADI_AD7879_CMD_SET_SPI_CS
TWI Controller and Device	(0 << 8) | 0x2Fu	Processor dependent	ADI_AD7879_CMD_SET_TWI_DEVICE_ADDRESSES

Table 4 – Default Settings

5.3. 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	Processor dependent	ADI_AD7879_CMD_SET_SPI_DEVICE_NUMBER
SPI Chipselect	1 to 7	ADI_AD7879_CMD_SET_SPI_CS

Table 5 – Additional Required Settings

6. Hardware Considerations

For AD7879:

- The client should set the SPI device number to be used to for AD7879 before accessing its registers. Command id 'ADI_AD7879_CMD_SET_SPI_DEVICE_NUMBER' can be used to set SPI device number for AD7879.

- The client should set the SPI chip-select value (configure SPI_FLG) corresponding to AD7879 before accessing the device registers. Command id 'ADI_AD7879_CMD_SET_SPI_CS' can be used to set AD7879's chip-select.

For both AD7879 and AD7879-1:

- To enable interrupt monitoring, client should connect the AD7879 to a Blackfin port and configure AD7879 driver to monitor this signal. Commands ADI_AD7879_CMD_ENABLE_INTERRUPT_PENIRQ, ADI_AD7879_CMD_ENABLE_INTERRUPT_DAV and ADI_AD7879_CMD_ENABLE_INTERRUPT_ALERT can be used to enable monitoring PENIRQ, DAV and ALERT interrupts respectively.

6.1. AD7879 registers

Please note that the AD7879 device driver handles the most common use case of a touch screen device, that being the reporting of the X and Y positions and the pressure measurement, Z1.

If you have requirements beyond this use case you will need to modify the driver sources accordingly. The AD7879 data sheet lists the registers on the AD7879. Register names and bit-field macros are defined in AD7879 driver header file (adi_ad7879.h). Applications must use these corresponding register / register field macro to access the corresponding AD7879 device register.

7. Using AD7879 Driver in Applications

This section explains how to use AD7879 device driver in an application.

7.1. Device Manager Data memory allocation

The application should allocate base memory + memory for one SPI / TWI device + memory for AD7879 device + memory for other devices used by the application.

7.2. Interrupt Manager Data memory allocation

The application should allocate secondary interrupt memory for one Blackfin interrupt monitoring one AD7879 interrupt (PENIRQ, DAV & ALERT).

7.3. Flag Manager Data memory allocation

The application should allocate memory for one Blackfin flag monitoring one AD7879 interrupt (PENIRQ, DAV or ALERT).

7.4. Typical usage of AD7879 driver

a. AD7879 (driver) initialization

Step 1: Open AD7879 Device driver with device specific entry point

For AD7879:

Step 2: Set SPI device number to be used for AD7879 register access

Step 3: Set SPI Slave select number to be used for AD7879 register access

For AD7879-1:

Step 2: Set TWI controller number and device address if defaults not appropriate.

Step 3: Set TWI initialisation command table if default not suitable.

For both AD7879 and AD7879-1:

Step 4: Enable AD7879 interrupt monitoring

b. Initializing and controlling AD7879 (hardware)

Step 5: Configure AD7879 device to specific operating mode using device access service

Step 6: Respond to AD7879 callbacks

c. Terminating AD7879 driver

Step 7: Disable AD7879 interrupt monitoring

Step 8: Terminate AD7879 driver with adi_dev_Terminate( )

7.5. Configuring AD7879 on ADSP-BF526 EZ-BOARD

Please refer to the example fie TouchScreenClass.cpp located in Blackfin\Examples\Landscape LCD EZ-EXTENDER\SketchPad.

7.6. Re-use/share the SPI device used by AD7879

On receiving register access commands, AD7879 driver opens the allocated SPI device, performs device read/write via SPI and closes the SPI device soon after completing the register access. So the application can reuse/share the same SPI device used by AD7879 device, without closing the AD7879 driver itself.

8. Data Structures used by AD7879 driver

8.1. ADI_DEV_ACCESS_REGISTER

```
/* Data structure used for Single and Selective Register Access
*/ /* Data structure to access a single register */
/* Array structure to access a table of selective device registers
*/ typedef struct ADI_DEV_ACCESS_REGISTER
{
        u16     Address;        /* Device register address */
        u16     Data;           /* Data read/written from/to the register */
} ADI_DEV_ACCESS_REGISTER;
```

8.2. ADI_DEV_ACCESS_REGISTER_BLOCK

```
/* Data structure to access a block of consecutive registers
*/ typedef struct ADI_DEV_ACCESS_REGISTER_BLOCK
{
        u32     Count;          /* number of registers to be accessed */
        u16     Address;        /* starting address of register block */
        u16     *pData;  /* pointer to a 'Count' sized array of register data read/written from/to the device */
} ADI_DEV_ACCESS_REGISTER_BLOCK;
```

8.3. ADI_DEV_ACCESS_REGISTER_FIELD

```
/* Data structure to access individual register fields */

/* Basic element to access single register field */
/* Array structure to access a table of selective device register(s) field(s) */
typedef struct ADI_DEV_ACCESS_REGISTER_FIELD
{
        u16 Address;            /* Register address to access */
        u16 Field;              /* Register field in the above address to be accessed */
        u16 Data;               /* Register field data read/written from/to the device */
} ADI_DEV_ACCESS_REGISTER_FIELD;
```

8.4. ADI_AD7879_INTERRUPT_PORT

```
/* Structure to set AD7879 PENIRQ/DAV/ALERT Interrupt Flag connected to Blackfin */

/*
To enable AD7879 PENIRQ or DAV or ALERT interrupt, client must pass corresponding interrupt enable command with pointer to
following structure as value. The 'FlagId' field should hold the Blackfin processor Flag ID connected to the selected Interrupt signal
and 'FlagIntId' field should hold the Peripheral Interrupt ID of the corresponding flag
 */

typedef struct
{
        ADI_FLAG_ID             FlagId;     /* Flag ID connected to AD7879 interrupt signal     */
        ADI_INT_PERIPHERAL_ID   FlagIntId;  /* Peripheral Interrupt ID of the corresponding flag*/
} ADI_AD7879_INTERRUPT_PORT;
```

8.5. ADI_AD7879_RESULT_REGS

Listed here is the structure as defined in the file Blackfin\include\drivers\touchscreen\adi_7879.h

```
/*
** ADI_AD7879_RESULT_REGS
**  - AD7879 Result Register structure
**    Pointer to this register structure will be passed as callback argument
**    for Sequencer Slave and Master Mode read request, provided that
**    the client configures the driver to monitor the
**    Data Available (INTIRQ) interrupt
*/
typedef struct __AdiAd7879ResultRegs
{

    /* Variable: nX - X position measurement (Y+ input) */
    u16 nX;

    /* Variable: nY - Y position measurement (X+ input) */
    u16 nY;

    /* Variable: nZ1 - Z1 measurement (X- input with X+ & Y- excited) */
    u16 nZ1;

    /* Variable: nZ2 - Z2 measurement (Y- input with Y+ & X- excited) */
    u16 nZ2;

    /* Variable: nAuxVbat - AUX/VBAT voltage measurement */
    u16 nAuxVbat;

    /* Variable: nTemperature - Temperature conversion measurement */
    u16 nTemperature;

}ADI_AD7879_RESULT_REGS;
```

9. Programming Examples

9.1. Reading AD7879 device registers

This section explains how to access the AD7879 device registers using device access commands. Refer to section 6.1 for a list of the AD7879 device registers and for register field definitions.

9.1.1. Read a single AD7879 device register

Command: ADI_DEV_CMD_REGISTER_READ
Value: ADI_DEV_ACCESS_REGISTER* (register specifics)

Example:

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

/* Load the register address to be read */
ReadReg.Address = AD7879_REG_CONTROL_1;

/* Application calls adi_dev_Control( ) function with corresponding command and value

   Register value will be read back to location - ReadReg.Data */
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_READ, (void *) &ReadReg);
```

9.1.2. Read a single AD7879 device register field

Command: ADI_DEV_CMD_REGISTER_FIELD_READ
Value: ADI_DEV_ACCESS_REGISTER_FIELD* (register specifics)

Example:

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

/* Load the device register address to be accessed */
ReadField.Address = AD7879_REG_CONTROL_1;
/* Load the device register field location to be read */
ReadField.Address = AD7879_RFLD_PENIRQEN;

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

9.1.3. Read a table of AD7879 device registers

Command: ADI_DEV_CMD_REGISTER_TABLE_READ
Value: ADI_DEV_ACCESS_REGISTER* (register specifics)

Example:

```
/* define the structure to access table of device registers */
ADI_DEV_ACCESS_REGISTER ReadRegs [ ] =
        {       { AD7879_REG_CONTROL_1,         0},
                { AD7879_REG_CONTROL_2,         0},
                {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 ReadRegs array
*/
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_READ, (void *) &ReadRegs[0]);
```

9.1.4. Read a table of AD7879 device register(s) Fields

Command: ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ
Value: ADI_DEV_ACCESS_REGISTER_FIELD* (register specifics)

Example:

```
/* define the structure to access table of device register(s) fields */
ADI_DEV_ACCESS_REGISTER_FIELD ReadFields [ ] =
        {       { AD7879_RFLD_PENIRQEN,     AD7879_GPIO1_DAT,     0},
                { AD7879_RFLD_CHADD,        AD7879_AUX1HI,        0},
                { ADI_DEV_REGEND,           0,                    0}                    };

/*
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 ReadFields table
*/
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ,
                        (void *) &ReadFields [0]);
```

9.1.5. Read a block of AD7879 device registers

Command: ADI_DEV_CMD_REGISTER_BLOCK_READ
Value: ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics)

Example:

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

/* load the number of registers to be read */
ReadBlock.Count = 5;
/* load the starting address of the register block to be read */
ReadBlock.Address = AD7879_REG_XPOS;
/* define a 'Count' sized array to hold register data read from the device
*/ u16 DataBlock[5] = { 0 };
/* load the start address of the above array to 'ReadBlock' data pointer */
ReadBlock.pData = &DataBlock[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 DataBlock[ ] array
   value of AD7879_YPOS will be read to DataBlock[0] and following 4 registers to remaining
   locations in DataBlock[] array */
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_BLOCK_READ, (void *) &ReadBlock);
```

9.2. Configuring AD7879 device registers

9.2.1. Configure a single AD7879 register

Command: ADI_DEV_CMD_REGISTER_WRITE
Value: ADI_DEV_ACCESS_REGISTER* (register specifics)

Example:

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

/* Load the register address to be configured */
CfgReg.Address = AD7879_REG_AUX_BAT_HLIMIT;
/* Load the configuration value to CfgReg.Data location
*/ CfgReg.Data = 0xEF;

/* Application calls adi_dev_Control( ) function with corresponding command and value. The device register will be configured with the value in CfgReg.Data */
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_WRITE, (void *) &CfgReg);
```

9.2.2. Configure a single AD7879 register field

Command: ADI_DEV_CMD_REGISTER_FIELD_WRITE
Value: ADI_DEV_ACCESS_REGISTER_FIELD* (register specifics)

Example:

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

/* Load the device register address to be accessed */
CfgField.Address = AD7879_REG_CONTROL_2;
/* Load the device register field location to be configured */
CfgField.Address = AD7879_RFLD_GPIO_EN;
/* load the new field value */
CfgField.Data = 1;

/* Application calls adi_dev_Control( ) function with corresponding command and
   value Selected register field will be configured to the given value */
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_WRITE, (void *) &CfgField);
```

9.2.3. Configure a table of AD7879 registers

Command: ADI_DEV_CMD_REGISTER_TABLE_WRITE
Value: ADI_DEV_ACCESS_REGISTER* (register specifics)

Example:

```
/* define the structure to configure table of device registers (register address, configuration value) */
ADI_DEV_ACCESS_REGISTER CfgRegs [ ] =
        {       {AD7879_REG_CONTROL_2,      0x2C },
                { AD7879_REG_CONTROL_1,     0xEF },
                {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 *) &CfgRegs [0]);


/* 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_FIELD_TABLE_WRITE,
                        (void *) &CfgFields [0]);
```

9.2.4. Configure a table of AD7879 register(s) fields

Command: ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE
Value: ADI_DEV_ACCESS_REGISTER_FIELD* (register specifics)

Example:

```
/* 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 CfgFields [ ] =
        {       { AD7879_REG_CONTROL_2, AD7879_RFLD_GPIO_DAT, 1 },
                { AD7879_REG_CONTROL_2, AD7879_RFLD_GPIO_EN, 1 },
                { ADI_DEV_REGEND, 0, 0 }
        };

/* 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_FIELD_TABLE_WRITE,
                        (void *) &CfgFields [0]);
```

9.2.5. Configure a block of AD7879 registers

Command: ADI_DEV_CMD_REGISTER_BLOCK_WRITE
Value: ADI_DEV_ACCESS_REGISTER_BLOCK* (register specifics)

Example:

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

/* load the number of registers to be configured */
CfgBlock.Count = 8;
/* load the starting address of the register block to be configured */
CfgBlock.Address = AD7879_REG_CONTROL_1;

/* define a 'Count' sized array to hold register data read from the device
   load the array with AD7879 register configuration values */
u16 CfgData [8] = {0xEF, 0x0E, 0xF0, 0x20, 0xE9, 0x10, 0xF2, 0x38};

/* load the start address of the above array to CfgData data pointer */
CfgBlock.pData = &CfgData[0];

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

9.3. Command to set AD7879 SPI Device Number

Command: ADI_AD7879_CMD_SET_SPI_DEVICE_NUMBER
Value: u8

Example:

```
adi_dev_Control (DriverHandle, ADI_AD7879_CMD_SET_SPI_DEVICE_NUMBER, (void *)AD7879_SPI_CS );
```

9.4. Command to set AD7879 SPI Chipselect

Command: ADI_AD7879_CMD_SET_SPI_CS
Value: u8

Example:

```
adi_dev_Control (DriverHandle, ADI_AD7879_CMD_SET_SPI_CS, (void *) AD7879_SPI_CS);
```

9.5. Commands to enable AD7879 interrupt monitoring

9.5.1. Enable PENIRQ Monitoring

Command: ADI_AD7879_CMD_ENABLE_INTERRUPT_PEN_INT_IRQ
Value: ADI_AD7879_INTERRUPT_PORT*

Example:

```
ADI_AD7879_INTERRUPT_PORT          PenIrqPort;
PenIrqPort.FlagId      = ADI_FLAG_yourIdHere;
PenIrqPort.FlagIntId = ADI_INT_yourINterruptIDHere;

/* Configure AD7879 driver to monitor PENIRQ interrupt */
adi_dev_Control (DriverHandle, ADI_AD7879_CMD_ENABLE_INTERRUPT_PENIRQ, (void *) &PenIrqPort );
```

9.6. Commands to Disable AD7879 interrupt monitoring

9.6.1. Disable PENIRQ Monitoring

Command: ADI_AD7879_CMD_DISABLE_INTERRUPT_PENIRQ
Value: NULL

Example:

```
/* Remove PENIRQ from monitoring */
adi_dev_Control (DriverHandle, ADI_AD7879_CMD_DISABLE_INTERRUPT_PENIRQ, (void *) NULL);
```

10. References

1. Analog Devices AD7879 Touch Screen Controller DataSheet, Rev B, July 2006 http://www.analog.com/UploadedFiles/Data_Sheets/AD7879.pdf