ANALOG DEVICES

ADI_MAX1233 DEVICE DRIVER

DATE: NOVEMBER 8, 2007

Table of Contents

1 Overview 5

2 Files 6

2.1 Include Files 6

2.2 Source Files 6

2.3 Example Files 6

3 Lower Level Drivers 7

3.1 SPI Device Driver 7

3.2 Device Access System Service 7

4 Resources Required 7

4.1 Interrupts 8

4.2 Deferred Callbacks (DCB) 9

4.3 DMA 9

4.4 Timers 9

4.5 Real-Time Clock 9

4.6 Programmable Flags 9

4.7 Pins 10

5 Supported Features of the Device Driver 10

5.1 Directionality 10

5.2 Dataflow Methods 10

5.3 Buffer Types 10

5.4 Data Structures 11

5.5 Command IDs 11

5.6 Callback Events 14

5.7 Return Codes 15

6 Opening and Configuring the Device Driver 17

6.1 Entry Point 17

6.2 Default Settings 18

6.3 Additional Required Configuration Settings 18

7 Hardware Considerations 19

7.1 MAX1233 Registers 19

7.2 MAX1233 Register Fields.....21

8 Using the MAX1233 Device Driver in Applications.....22

8.1 Data Memory Allocation.....22

8.2 Typical Usage of the MAX1233 Device Driver.....23

8.3 Configuring the MAX1233 Device Driver on the ADSP-BF527 EZ-Kit Lite Evaluation Board.....26

9 Programming Example.....27

9.1 ApplicationCallback Parameters.....27

9.2 Processing the Event.....28

9.3 Sync Output.....28

9.4 Test A Status Bit.....29

9.5 Run ADC.....29

9.6 Read Conversion Data.....30

9.7 Return ADC to Mode1.....30

9.8 Test Pen Hold Status.....30

9.9 Dummy Read.....31

9.10 Re-Enable Interrupts.....31

List of Tables

Table 1 – Revision History.....4

Table 2 – Supported Dataflow Directions.....10

Table 3 – Supported Dataflow Methods.....10

Table 4 – Default Settings.....18

Table 5 – Additional Required Settings.....18

Table 6 – Max1233 Page Zero Read-Only Data Registers.....19

Table 7 – Max1233 Page Zero Read-Only Data Registers.....20

Table 8 – MAX1233 Register Fields.....21

Document Revision History

Date	Description of Changes
11/8/07	Initial Release

Table 1 – Revision History

1 Overview

This Device Driver provides an effective and easy way to manage the Maxim Integrated Products, Inc., MAX1233 PDA Combined Touch-Screen and Keypad Controller integrated circuit (IC). This Device Driver is developed and deployed on the ADSP-BF527 EZ-Kit, which incorporates the MAX1233 controller IC to manage the Varitronix Touch-Screen (part of the combination Touch-Screen/LCD display module) and generic Keypad encoder which are also part of the ADSP-BF527 EZ-Kit Lite. Other hardware platforms may differ in implementation, but the MAX1233 Device Driver should accommodate them easily, provided the configuration settings are adjusted accordingly.

The MAX1233 Device Driver provides complete communications between the MAX1233 IC and the ADSP-BF527 processor over a shared Serial Peripheral Interface (SPI). The MAX1233 IC operates as the SPI slave device and the ADSP-BF527 processor operates as the SPI master device. The Device Driver allows high-level management of MAX1233 configuration, data acquisition subsystems and the SPI channel itself. The Device Driver hides the low-level SPI tasks, pin configuration, chip select, interrupt configuration, register mapping, etc., all of which greatly simplifies the higher-level Application coding requirements to deploy the MAX1233 controller quickly and easily.

The MAX1233 Device Driver also manages interrupt handling of the two MAX1233 interrupt signals (PENIRQ, KEYIRQ) by allowing registration of and dispatching interrupt Callbacks to the Application Layer.

Working knowledge of the Maxim MAX1233/MAX1234 data sheet, “MAXIM +/-15kV ESD-Protected Touch-Screen Controllers Include DAC and Keypad Controller”, is assumed throughout this document. Please refer to the manufactures’ data sheet for a complete description and specifications for this product: http://datasheets.maxim-ic.com/en/ds/MAX1233-MAX1234.pdf.

2 Files

The files listed below comprise the Device Driver API, implementation source and example files.

2.1 Include Files

The Device Driver sources references 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/spi/adi_spi.h>
- This file contains all definitions, function prototypes etc. for the SPI System Service.

<drivers/deviceaccess/adi_device_access.h>
- This file contains all definitions, function prototypes etc. for the Device Access System Service, used to encapsulate the SPI communications protocol.

<drivers/touchscreen/maxim/adi_max1233.h>
- This file contains all definitions, function prototypes etc. specific to the MAX1233 controller.

2.2 Source Files

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

< Blackfin/lib/src/drivers/touchscreen/maxim/adi_max1233.c>
- This file contains all the source code for the MAX1233 Device Driver. All source code is written in 'C'. There are no assembly level functions in this Device Driver. Include this source file in the build of the MAX1233 Application code base.

2.3 Example Files

This Device Driver is shipped with the following Example Files demonstrating use of the MAX1233 Device Driver. The entire example is based on the hardware platform provided by the "ADSP-BF527 Ez-Kit Lite" evaluation board. The Example is discussed in detail in later sections of this document.

<Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\touchscreen-keypad\adi_ssl_Init.c>
- This file contains example-specific System Service initialization source code.

<Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\touchscreen-keypad\asi_ssl_Init.h>
- This file contains example-specific System Service initialization include code.

<Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\touchscreen-keypad\Max1233Example.c>
- This file contains a complete application program, demonstrating use of the MAX1233 Device Driver to control the TouchScreen and Keypad hardware resources on the "ADSP-BF527 EZ_Kit Lite" evaluation board.

<Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\touchscreen-keypad\Max1233Example.dpj>
- This file contains a Visual DSP++ 5.0 IDDE project file to build the example Application.

<Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\touchscreen-keypad\Readme_Max1233Example.txt>
- This file contains the README.TXT material.

3 Lower Level Drivers

MAX1233 Device Driver is layered on top of the Device Access Service, which completely encapsulates the SPI and TWI serial communication protocols. Direct management of the SPI System Service is not required, either in the Application Layer or in the MAX1233 Device Driver.

3.1 SPI Device Driver

The MAX1233 controller is accessed over an SPI-based serial communications channel. All control and data functions are implemented by reading from and writing to MAX1233 registers via the SPI interface. These registers are addressed through a 16-bit command word that is sent prior to the data. The command word includes a read/write bit, a page select bit and six register address bits. The read/write data follows the command word. The details of the command word format, register map, chip-select and interrupt timing are contained in the MAX1233 data sheet.

3.2 Device Access System Service

The SPI System Service implements the low-level SPI communications with the MAX1233. The SPI System Service is wrapped by the high-level Device Access Service serial communications protocol. The Device Access System Service provides a singular interface that encapsulates both the SPI and TWI serial communications protocols under a common API. All SPI communications with the MAX1233 IC are managed by the Device Access System Service (via the MAX1233 Device Driver) and the Application Layer need not interact with the underlying SPI Device Driver at all.

The Device Access System Service also allows the SPI Device Driver to be shared by other resources while it is not actively communicating with the MAX1233 IC. The Device Access System Service manages the underlying SPI Device Driver resources by allocating, using and releasing them on each access, so that the SPI Device Driver is kept free for other use. SPI configuration parameters unique to MAX1233 access are stored within the MAX1233 Device Driver during driver initialization.

The Applications Layer and the MAX1233 Device Driver do not make direct use of the underlying SPI System Service. Please read the separate Device Access System Service manual for further details.

4 Resources Required

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

Unless explicitly noted in the sections below, this Device Driver uses the System Services to access and control any required hardware. The information in this section may be helpful in determining the resources this Device 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 allocations for the Device Manager and System Services are declared in the System Services Library initialization file, adi_ssl_Init.c, which is typically localized to the needs of the Application.

Wherever possible, this Device Driver uses the System Services to perform the necessary low-level hardware access and control.

This Device Driver is built on top of the Device Access System Service and the interrupt-driven, SPI Device Driver. **Each MAX1233 device requires two additional memory pools of size ADI_DEV_DEVICE_MEMORY, one for MAX1233 and one for SPI.**

4.1 Interrupts

The MAX1233 Device Driver is capable of processing two MAX1233 IC interrupt signals: the Pen Interrupt (PENIRQ) and the Key Interrupt (KEYIRQ). These interrupt signals are mapped through Blackfin GPIO ports as flag inputs. It is the Application's responsibility to provide additional, secondary interrupt memory to accommodate these flag interrupts, as well as to register a Callback handler which will receive callback events from the MAX1233 Device Driver.

Upon receipt of a properly configured and enabled interrupt, the Interrupt System Service makes a Callback into the MAX1233 Device Driver, which in turn, dispatches a unique event ID to the Application Layer Callback routine.

The Application Layer Callback can be either "deferred" (if "deferred Callbacks" are enabled) or "live" (if "deferred Callbacks" are disabled). The distinction is that "live" Callbacks are dispatched by a *direct call* to the Applications Callback routine while the ADSP-BF527 is *still in the interrupt service routine*; with the result that further interrupt dispatching of equal or lower-priority interrupts is blocked. In contrast, "deferred" Callbacks are dispatched by a *posted call* which is made from the Interrupt System Service *after the associated interrupt service routine is exited*. (See the Interrupt System Service and Deferred Callback System Service documentation for details.)

Depending on the mode of operation, the MAX1233 controller IC may issue multiple interrupt requests as user code responds to the initial interrupt. Each ADC conversion command and corresponding data read command can generate an additional interrupt. Internally, the MAX1233 IC blocks further interrupts until any pending conversion data is read. If the pen/key is still down as each result data register is read, another interrupt request will be issued. This can lead to many interrupts being generated in response to a complex operator sequence such as touch/drag/drop or key-press and hold.

The MAX1233 Device Driver is designed to mask multiple interrupts by disabling the interrupt input on the ADSP-BF527 processor as the initial interrupt is dispatched to the Application Callback, thereby blocking subsequent interrupts from the same source until the Application Layer tells the MAX1233 Device Driver to re-enable them. In essence, the MAX1233 takes a "lock-step" approach to interrupt dispatching by gating interrupts off (after the first one in a possible sequence) until the Application instructs it to allow them again. The Application re-enables interrupt dispatching at the conclusion of each "event" processing by issuing one of the corresponding:

ADI_MAX1233_CMD_REENABLE_PENIRQ, or
ADI_MAX1233_CMD_REENABLE_KEYIRQ

commands.

This Device Driver requires additional memory for two flag interrupts, where each interrupt requires 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.

4.2 Deferred Callbacks (DCB)

The MAX1233 Device Driver is capable of servicing the two interrupt signals: the Pen Interrupt (PENIRQ) and the Key Interrupt (KEYIRQ) either *during* the interrupt (immediate) or *outside* the interrupt (deferred). The distinction is whether the Device Driver interrupt Callback does a call (from the ISR) or a post to the Application Callback. This is discussed above.

Deferred Callbacks are preferred for quick interrupt response time (i.e., interrupt-level processing is kept to a minimum); while immediate (non-deferred or "live") Callbacks are preferred for low overhead and critical or real-time processing that demands immediacy. Use of Deferred Callbacks is a user choice made in the Application Layer.

The example code provided with the MAX1233 Device Driver controls the use of deferred Callbacks by asserting the USE_DEFERRED_CALLBACKS macro. If using DCB, the Application is responsible for providing additional, secondary, interrupt memory to accommodate these flag interrupts. **This Device Driver requires additional memory for two DCB servers, one for each Callback (in response to PENIRQ and KEYIRQ), where each DCB service requires memory of size ADI_DCB_QUEUE_SIZE.**

4.3 DMA

MAX1233 Device Driver does not use or support direct use of DMA. The SPI System Service does use DMA "under the hood" for managing data transfers within the SPI hardware, but this is entirely hidden from the MAX1233 Device Driver and the MAX1233 Application Layer. No DMA resource allocations are required at the Application or Device Driver Layers.

4.4 Timers

Timers are not used by this Device Driver.

4.5 Real-Time Clock

Real-time clock is not used by this Device Driver.

4.6 Programmable Flags

The MAX1233 is capable of generating two interrupt signals: the Pen Interrupt (PENIRQ) and the Key Interrupt (KEYIRQ). These interrupt signals are mapped through Blackfin GPIO ports (flags) which require configuration.

The MAX1233 Device Driver can be configured to monitor any or all the above interrupt signals. It is the Application's responsibility to initialise the flag service (using adi_flag_Init( )) with enough memory for flag Callbacks. **This Device Driver can monitor both MAX1233 interrupt signals and each flag connected to MAX1233 interrupt requires memory of size ADI_FLAG_CALLBACK_MEMORY to manage its Callback.**

4.7 Pins

Connect the desired Blackfin SPI port pins to the MAX1233 SPI port (note: the ADSP-BF527 processor offers two different port mappings of the SPI pins; see the “ADSP-BF52x Blackfin Processor Hardware Reference” manual). The pin selection is conveyed to the MAX1233 Device Driver through configuration commands which include the pin and the associated interrupt mapping.

The MAX1233 IC “X” and “Y” force and sense lines are connected to the touch screen and if a keypad is used, the row and column pins are connected to it. Other optional connections are in support of external inputs for monitoring external battery voltages, auxiliary analog inputs, an external ADC reference voltage, a DAC analog output and provision for a GPIO interface in lieu of a Keypad connection.

5 Supported Features of the Device Driver

This section describes what features are supported by the Device Driver.

5.1 Directionality

The Device 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 Device Driver supports the dataflow methods listed in the table below.

ADI_DEV_MODE	Description
ADI_DEV_MODE_CHAINED	Supports the chained buffer method

Table 3 – Supported Dataflow Methods

5.3 Buffer Types

The MAX1233 Device Driver doesn’t support buffer submission functions like adi_dev_Read(), adi_dev_Write() and adi_dev_SequentialIO() and returns the ADI_MAX1233_RESULT_CMD_NOT_SUPPORTED result code for these calls. The Application uses the adi_dev_Control() function exclusively to access all MAX1233 device control and data registers. Refer to the example Application.

5.4 Data Structures

The MAX1233 Device Driver uses the MAX1233_INTERRUPT_PORT data structure to pass interrupt configuration information from the Application Layer to the Device Driver when setting up interrupt Callbacks for either of the ADI_MAX1233_CMD_INSTALL_PENIRQ or ADI_MAX1233_CMD_INSTALL_KEYIRQ commands. This is a required step in configuring the MAX1233 Device Driver. The typedef is defined as:

```
typedef struct {
  ADI_FLAG_ID           FlagId;    /* Flag ID connected to MAX1233 interrupt signal     */
  ADI_INT_PERIPHERAL_ID FlagIntId; /* Peripheral Interrupt ID of the corresponding flag */
} MAX1233_INTERRUPT_PORT;
```

where ADI_FLAG_ID is an enumeration type defined in the Flag System Service include file (Blackfin\include\services\flag\adi_flag.h) and ADI_INT_PERIPHERAL_ID is an enumeration type defined in the Interrupt System Services include file (Blackfin\include\services\int\adi_int.h).

5.5 Command IDs

This section enumerates the commands that are supported by the Device 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 Device Driver supports. The remaining section describes Device 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 Device Driver and the meaning of the Value parameter for each command ID.

5.5.1 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 Device 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 *

The MAX1233 Device Driver intercepts the following Device Manager commands. In the case of dataflow commands, the MAX1233 simply returns without doing anything, as the underlying SPI Device Access communications protocol requires the ADI_DEV_ DIRECTION_BIDIRECTIONAL dataflow mode exclusively. In the case of DMA support query, the MAX1233 Device Driver returns FALSE, as it does not support DMA.

ADI_DEV_CMD_SET_DATAFLOW
- Attempts to modify the dataflow are ignored
- Value – disregarded

ADI_DEV_CMD_SET_DATAFLOW_METHOD
- Attempts to query the dataflow are ignored
- Value – disregarded

ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT
- Queries whether the Device Driver supports DMA
- Value – pointer to a u32 result, which is set to “FALSE” on return

5.5.2 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. These commands are passed to the underlying Device Access Device Driver for execution.

DEVICE DRIVER ERATTA: Note that “field” and “block” access methods (ADI_DEV_CMD_REGISTER_FIELD_READ, ADI_DEV_CMD_REGISTER_BLOCK_READ, etc.) are not supported on the initial release of the MAX1233 Device Driver. Please use complete register read/write commands (ADI_DEV_CMD_REGISTER_READ, ADI_DEV_CMD_REGISTER_TABLE_READ, etc.) of fully-formed register values. This is demonstrated in the example.

ADI_DEV_CMD_REGISTER_READ
- Reads a single device register
- Value – ADI_DEV_ACCESS_REGISTER * (register specifics)

ADI_DEV_CMD_REGISTER_FIELD_READ
- Reads a specific field location in a single device register
- Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)

ADI_DEV_CMD_REGISTER_TABLE_READ
- Reads a table of selective device registers
- Value – ADI_DEV_ACCESS_REGISTER * (register specifics)

ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ
- Reads a table of selective device register fields
- Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)

ADI_DEV_CMD_REGISTER_BLOCK_READ
- Reads a block of consecutive device registers
- Value – ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics)

ADI_DEV_CMD_REGISTER_WRITE
- Writes to a single device register
- Value – ADI_DEV_ACCESS_REGISTER * (register specifics)

ADI_DEV_CMD_REGISTER_FIELD_WRITE
- Writes to a specific field location in a single device register
- Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)

ADI_DEV_CMD_REGISTER_TABLE_WRITE
- Writes to a table of selective device registers
- Value – ADI_DEV_ACCESS_REGISTER * (register specifics)

ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE
- Writes to a table of selective device register fields
- Value – ADI_DEV_ACCESS_REGISTER_FIELD * (register specifics)

ADI_DEV_CMD_REGISTER_BLOCK_WRITE
- Writes to a block of consecutive device registers
- Value – ADI_DEV_ACCESS_REGISTER_BLOCK * (register specifics)

5.5.3 Device Driver Specific Commands

The command IDs listed below are supported and processed by the MAX1233 Device Driver. These command IDs are unique to this Device Driver. These MAX1233 Device Driver command IDs are defined in the "adi_max1233.h" file. See example illustrating use of these commands.

ADI_MAX1233_CMD_SET_SPI_DEVICE_NUMBER
- Sets Blackfin SPI Device Number to be used to access MAX1233 registers (Refer to example)
- Value – u8

ADI_MAX1233_CMD_SET_SPI_CS
- Sets Blackfin SPI Slave Select (SPI_SSEL#) used to drive the chip select input on the MAX1233 device for SPI access (Refer to example)
- Value – u8

ADI_MAX1233_CMD_SET_SPI_SLAVE
- Deprecated command – do not use. Activating the SPI SLAVE mode and the corresponding SPI Chip Select commands are combined under the singular ADI_MAX1233_CMD_SET_SPI_CS command. This command will return ADI_MAX1233_RESULT_CMD_NOT_SUPPORTED in future releases.
- Value – Ignored

ADI_MAX1233_CMD_SET_SPI_BAUD
- Sets the internal ADSP-BF527 SPI_BAUD baud rate register value through the Device Access Device Driver. This command controls the SPI clock frequency as SCLK / (2*SPI_BAUD). See example.
- Value – u32* (pointer to the baud rate value)

ADI_MAX1233_CMD_INSTALL_PENIRQ
- Configures flag and interrupt selection for the PENIRQ input.
- Value – MAX1233_INTERRUPT_PORT*, a pointer to a `MAX1233_INTERRUPT_PORT` data structure containing the FlagID and InterruptID to use for the PENIRQ interrupt

ADI_MAX1233_CMD_INSTALL_KEYIRQ
- Configures flag and interrupt selection for the KEYIRQ input.
- Value – MAX1233_INTERRUPT_PORT*, a pointer to a `MAX1233_INTERRUPT_PORT` data structure containing the FlagID and InterruptID to use for the KEYIRQ interrupt

ADI_MAX1233_CMD_UNINSTALL_PENIRQ
- Uninstalls PENIRQ handling
- Value – NULL

ADI_MAX1233_CMD_UNINSTALL_KEYIRQ
- Uninstalls KEYIRQ handling
- Value – NULL

ADI_MAX1233_CMD_REENABLE_PENIRQ
- Re-Enables MAX1233 Device Driver to dispatch PENIRQ interrupt requests after the Application receives the ADI_MAX1233_EVENT_PENIRQ_NOTIFICATION event
- Value – ADI_MAX1233_INTERRUPT_PORT*

ADI_MAX1233_CMD_REENABLE_KEYIRQ
- Re-Enables MAX1233 Device Driver to dispatch KEYIRQ interrupt requests after the Application receives the ADI_MAX1233_EVENT_KEYIRQ_NOTIFICATION event
- Value – ADI_MAX1233_INTERRUPT_PORT*

5.6 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 Device 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.

5.6.1 Common Events

There are no common event IDs supported by this Device Driver

5.6.2 Device Driver Specific Events

The events listed below are generated by the Device Driver. These event IDs are unique to this Device Driver. These MAX1233 Device Driver event IDs are defined in the “adi_max1233.h” file.

Two event notifications are possible, in response to receipt of either the PENIRQ or KEYIRQ interrupts from the MAX1233 controller. Prior to dispatching the event notification to the Application Callback, the MAX1233 Device Driver disables subsequent (multiple) interrupts from the same source until the Application Layer explicitly re-enables them from the Application Callback (with either ADI_MAX1233_CMD_REENABLE_PENIRQ or ADI_MAX1233_CMD_REENABLE_KEYIRQ, as appropriate). The Application typically re-enables interrupting after processing of potentially extended sequences of events (e.g., “drag and drop”, or “key press and hold”) is complete.

The reason for gating off multiple interrupts is to accommodate multiple conversions and data reads in response to the same event, i.e., reading multiple results (possibly, pen position, pen pressure, temperature, battery voltage, key press code, etc.) or a sequence of actions (possibly, “drag and drop”, or “key press and hold”) within the same Application Callback. This is important in the case of Deferred CallBacks (DCB), which will otherwise result in the Application Callback being dispatched many times as the MAX1233 IC issues multiple interrupts with each data conversion/data read sequence. In the case of “live” interrupts (DCB disabled), multiple interrupts from the same interrupt source while the interrupt service routine is still active are already masked by the ADSP-BF527 Blackfin interrupt controller.

Whether using DCB or using “live” Callbacks, the respective interrupt *must* be re-enabled by the Application Callback in response to each event notification.

ADI_MAX1233_EVENT_PENIRQ_NOTIFICATION
- Indicates the PENIRQ interrupt has been detected, corresponding to a screen touch event. Re-enable this interrupt with ADI_MAX1233_CMD_REENABLE_PENIRQ.
- Value – NULL. Not used.

ADI_MAX1233_EVENT_KEYIRQ_NOTIFICATION
- Indicates the KEYIRQ interrupt has been detected, corresponding to a keypad press event. Re-enable this interrupt with ADI_MAX1233_CMD_REENABLE_KEYIRQ.
- Value – NULL. Not used.

5.7 Return Codes

All API functions of the Device Driver return status information indicating either successful completion of the function or 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 non-zero.

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 Device 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
}
```

5.7.1 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 Device 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 Device 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 Device Driver

5.7.2 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. These MAX1233 Device Driver return codes are defined in the “adi_max1233.h” file.

ADI_MAX1233_RESULT_CMD_BAD_BAUD_RATE
- The client attempted to set an invalid baud rate for SPI communications between this Device Driver and the target hardware.

ADI_MAX1233_RESULT_CMD_NOT_SUPPORTED
- Occurs when client issues a command which is not supported by this Device Driver

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 Device Driver is listed below.

ADIMAX1233EntryPoint

6.2 Default Settings

Table 4 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.

Please refer to the MAX1233 data sheet for possible values of control registers. Many of the Blackfin resource settings are ADSP-BF527-specific and vary is utility and range from processor to processor.

Item	Default Value	Possible Values	Command ID
MAX1233 ADC Control Register	0	See MAX1233 Data Sheet	Use Device Access commands to read/write this register
MAX1233 Key Control Register	0	See MAX1233 Data Sheet	Use Device Access commands to read/write this register
MAX1233 DAC Control Register	0x8000	See MAX1233 Data Sheet	Use Device Access commands to read/write this register
MAX1233 GPIO Pullup Register	0	See MAX1233 Data Sheet	Use Device Access commands to read/write this register
MAX1233 GPIO Control Register	0	See MAX1233 Data Sheet	Use Device Access commands to read/write this register
MAX1233 Key Mask Register	0	See MAX1233 Data Sheet	Use Device Access commands to read/write this register
MAX1233 Key Column Register	0	See MAX1233 Data Sheet	Use Device Access commands to read/write this register
SPI Device Number	0	0 or 1	ADI_MAX1233_CMD_SET_SPI_DEVICE_NUMBER
SPI Chip Select	0	1 through 7 (corresponding to use of SPISSEL1 through SPISSEL7)	ADI_MAX1233_CMD_SET_SPI_CS
SPI Baud Rate	0	2 through 65,535 (0 and 1 are invalid)	ADI_MAX1233_CMD_SET_SPI_BAUD

Table 4 – Default Settings

6.3 Additional Required Configuration Settings

In addition to the possible overrides of the default Device Driver settings in Table 4, the MAX1233 Device Driver requires the Application to specify the additional configuration information listed in Table 5.

Item	Possible Values	Command ID
Install PENIRQ Callback	MAX1233_INTERRUPT_PORT*	ADI_MAX1233_CMD_INSTALL_PENIRQ
Install KEYIRQ Callback	MAX1233_INTERRUPT_PORT*	ADI_MAX1233_CMD_INSTALL_KEYIRQ

Table 5 – Additional Required Settings

7 Hardware Considerations

The client Application should perform the required Device Driver and device configurations outlined above under "Opening and Configuring the Device Driver". The following descriptions appear in the MAXIM MAX1233 data sheet and are repeated here for convenience. Please reference this data sheet for complete details of all registers and bitfield descriptions and control modes for using the MAX1233.

7.1 MAX1233 Registers

The MAX1233 register map is divided into two pages. Page zero contains read-only data registers and page one registers contain read/write control registers. Conversion results and key-press information are stored in the page zero registers.

Each register page also contains various "Reserved" registers, whose use is undocumented by Maxim.

The MAX1233 Device Driver include file, <Blackfin\include\drivers\touchscreen\adi_max1233.h>, provides macros for all the register addresses, bitfield positions and bitfield values.

7.1.1 Page Zero Read-Only Data Registers

Table 6 shows the 12-bit ADC conversion results of various analog inputs. The data of these registers is right-justified at bit position 0. Data written to page-zero registers is not stored, unless it is for the properly configured DAC or GPIO registers.

Register	Address	Default	Description
X	0x0000	-N/A-	X-coordinate of touch-position
Y	0x0001	-N/A-	Y-coordinate of touch-position
Z1	0x0002	-N/A-	Z1 touch-pressure (X+ measurement with X- & Y+ forced)
Z2	0x0003	-N/A-	Z2 touch-pressure (Y- measurement with X- & Y+ forced)
KPD	0x0004	-N/A-	Keypad scan result (raw)
BAT1	0x0005	-N/A-	Voltage measurement of external BAT1 analog input
BAT2	0x0006	-N/A-	Voltage measurement of external BAT2 analog input
AUX1	0x0007	-N/A-	Voltage measurement of external AUX1 analog input
AUX2	0x0008	-N/A-	Voltage measurement of external AUX2 analog input
TEMP1	0x0009	-N/A-	Single-ended measurement of internal chip temperature
TEMP2	0x000A	-N/A-	Differential measurement of internal chip temperature
DAC	0x000B	-N/A-	8-bit Digital-to-analog converter input value
GPIO	0x000F	-N/A-	General-purpose data register for alternate use of external row and column pins
KPData1	0x0010	-N/A-	Keypad scan result (pending)
KPData2	0x0011	-N/A-	Keypad scan result (status)

Table 6 – Max1233 Page Zero Read-Only Data Registers

7.1.2 Page One Read/Write Control Registers

Table 7 shows the various control registers on the MAX1233. Most of these registers are bit-fielded for control of various modes, resolutions, and other conversion parameters. The bitfields are defined in the next section.

Table 7 – Max1233 Page Zero Read-Only Data Registers

Register	Address	Default	Description
ADC	0x0040	-N/A-	ADC control register
KEY	0x0041	-N/A-	Keypad control register
DAC	0x0042	-N/A-	DAC control register
GPIOPullup	0x004E	-N/A-	GPIO pullup disable register
GPIO	0x004F	-N/A-	GPIO control register
KPKeyMask	0x0050	-N/A-	Keypad key mask control register
KPColumnMask	0x0051	-N/A-	Keypad column mask control register

7.2 MAX1233 Register Fields

Table 8 shows the various control register bitfields on the MAX1233. Some of these bitfields are dual function, depending on whether the access is read or write; these fields are prefixed with an asterisk (*). Many fields are multi-bit and are designated FUNCTION#, where # is typically 0, 1, 2, etc. Please refer to the MAX1233 data sheet for complete descriptions of these bitfields.

Field	Position	Size	Description
ADC			
*PENSTS	15	1	Pen interrupt status
*ADSTS	14	1	ADC status
A/D#	10	4	Selects ADC scan functions
RES#	8	1	Controls ADC resolution
AVG#	6	2	Controls ADC result averaging
CNR#	4	2	Controls ADC conversion rate
ST#	1	3	Controls touch-screen settling wait time
RFV	0	1	Chooses 1.0 or 2.5 internal reference voltage
Keypad Control Register (KEY)			
*KEYSTS1	15	1	Keypad interrupt status
*KEYSTS0	14	1	Keypad scan status
DBN#	11	3	Keypad debounce time control
HLD#	8	3	Keypad hold time control
DAC			
DAPD	15	1	DAC power-down bit
GPIOPullup			
PU#	8	8	Controls internal pull-ups of GPIO outputs
GPIO			
GP#	8	8	Maps individual Column and Row pins as GPIO
OE#	0	8	Maps individual GPIO pin direction
KPKeyMask			
KM#	0	16	Mask status register data update in individual keys
KPColumnMask			
CM#	12	4	Mask interrupt,, status register and pending register data update on all keys in column

Table 8 – MAX1233 Register Fields

8 Using the MAX1233 Device Driver in Applications

This section explains how to use the MAX1233 Device Driver in an Application. The code fragments shown in the various sample declarations are taken from the Example source code. Individual applications may vary in memory and resource requirements.

8.1 Data Memory Allocation

The following resource defines are asserted in the example code, <InstallRoot>\Analog Devices\VisualDSP 5.0\Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\touchscreen-keypad\adi_ssl_init.h. Use of these macros is illustrated in the following Data Memory Allocation sections.

```
#define ADI_SSL_DEV_NUM_DEVICES            (2) // number of Device Drivers (MAX1234 & SPI)
#define ADI_SSL_INT_NUM_SECONDARY_HANDLERS (2) // number of secondary interrupt handlers (PENIRQ & KEYIRQ)
#define ADI_SSL_DCB_NUM_SERVERS            (2) // number of DCB servers
#define ADI_SSL_DMA_NUM_CHANNELS           (0) // number of DMA channels
#define ADI_SSL_FLAG_NUM_CALLBACKS         (2) // number of flag Callbacks (PENIRQ & KEYIRQ)
#define ADI_SSL_SEM_NUM_SEMAPHORES         (0) // number of semaphores
```

8.1.1 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 the MAX1233 device + memory for other devices used by the application (i.e., SPI). A sample allocation is illustrated as:

```
static u8 DevMgrData [ADI_DEV_BASE_MEMORY + (ADI_DEV_DEVICE_MEMORY * ADI_SSL_DEV_NUM_DEVICES)];
```

8.1.2 Interrupt Service Data Memory Allocation

This section explains Interrupt Manager memory allocation requirements for applications using this driver. The application should allocate secondary interrupt memory for two Blackfin flags monitoring MAX1233 interrupts (PENIRQ and KEYIRQ). A sample allocation is illustrated as:

```
static u8 InterruptServiceData [ADI_INT_SECONDARY_MEMORY * ADI_SSL_INT_NUM_SECONDARY_HANDLERS];
```

8.1.3 Deferred Callback Service Memory Allocation

This section explains Deferred Callback Manager memory allocation requirements for applications using this driver. If using DCB, the application should allocate sufficient memory for two interrupt callbacks (PENIRQ and KEYIRQ). A sample allocation is illustrated as:

```
static u8 DeferredCallbackServiceData [ADI_DCB_QUEUE_SIZE * ADI_SSL_DCB_NUM_SERVERS];
```

8.1.4 DMA Service Data Memory Allocation

This section explains DMA Manager memory allocation requirements for applications using this driver. The MAX1233 does not use DMA resources directly, though DMA is used by the SPI Device Manager. The application should allocate sufficient memory for whatever DMA resources are required. A sample allocation is illustrated as:

```
static u8 DMAServiceData [ADI_DMA_BASE_MEMORY + (ADI_DMA_CHANNEL_MEMORY *  ADI_SSL_DMA_NUM_CHANNELS)];
```

8.1.5 Flag Service Data Memory Allocation

This section explains Flag Manager memory allocation requirements for applications using this driver. The application should allocate memory for two Blackfin flags monitoring MAX1233 interrupts (PENIRQ and KEYIRQ). A sample allocation is illustrated as:

```
static u8 FlagServiceData [ADI_FLAG_CALLBACK_MEMORY * ADI_SSL_FLAG_NUM_CALLBACKS];
```

8.1.6 Semaphore Service Data Memory Allocation

This section explains Semaphore Manager memory allocation requirements for applications using this driver. The MAX1233 does not use Semaphore resources. The application should allocate sufficient memory for however many semaphores are required. A sample allocation is illustrated as:

```
static u8 SemaphoreServiceData [ADI_SEM_SEMAPHORE_MEMORY * ADI_SSL_SEM_NUM_SEMAPHORES];
```

8.2 Typical Usage of the MAX1233 Device Driver

See the Example code provided with the MAX1233 resides in the <InstallRoot>\Analog Devices\VisualDSP 5.0\Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\touchscreen-keypad directory. This example provides a complete example using the MAX1233 Device Driver. The sequence of operations described here is from the Applications Layer perspective and typifies the MAX1233 Device Driver usage.

Please refer to the appropriate System Services Library and Device Driver documentation for descriptions of generic data structures and handles used by the MAX1233 Device Driver and Example code.

8.2.1 Initialize the System Services Library

Call adi_ssl_Init() to perform generic System Services initialization according to the memory/resource allocations outlined above.

8.2.2 Open the DCB Manager

This is an optional step if the Application Layer is going to use the DCB (Deferred Callback) Service. The DCB manager handle thus obtained is passed to adi_dev_Open().

8.2.3 Open/Configure Any Flag Resources

Perform any sequence of adi_flag_Open(), adi_flag_SetDirection(), adi_flag_Set(), adi_flag_Clear(), etc., desired for configuring hardware Flags on the ADSP-BF527 processor for ancillary functions such as pushbuttons, logic analyzer sync bits, LEDs, etc.

Flag configurations in support of MAX1233 interrupts are managed separately as MAX1233_INTERRUPT_PORT (struct) pairings of flag and interrupt IDs, which are passed to the MAX1233 Device Driver during hardware initialization. See section “Configure the MAX1233 Device Driver” for details of these and other hardware configuration commands.

8.2.4 Query the Power System Service for SCLK

Call adi_pwr_GetFreq() to obtain the current (variable) system clock (SCLK) frequency so as to compute the desired baud rate for running the SPI channel.

The SPI baud rate is set as a function of the desired SPI clock frequency, thus:

SpiBaudRateRegValue = fsclk / (2*MAX1233_SPI_CLOCK_FREQ);

The resultant baud rate is sent to the MAX1233 Device Driver with the “ADI_MAX1233_CMD_SET_SPI_BAUD” command during MAX1233 Device Driver configuration.

8.2.5 Open the MAX1233 Device Driver

Call adi_dev_Open() and pass down the following parameters:

- Device Manager Handle
- MAX1233 Entry Point
- MAX1233 Device Number
- Client Handle
- MAX1233 Device Handle
- Data Directionality
- DMA Manager Handle
- DCB Manager Handle
- Application Callback Address

8.2.6 Configure the MAX1233 Device Driver

Once the Device Driver is successfully opened, a series of adi_dev_Control() calls are made to configure the Device Driver for service. Each adi_dev_Control() call conveys the MAX1233 Device Handle (obtained from adi_dev_Open()), a command identifier, and some command data.

The command identifier is typically one of the following. See section “Command IDs” for a complete list of Device Driver commands.

8.2.6.1 ADI_DEV_CMD_TABLE

A command table is an array of ADI_DEV_CMD_VALUE_PAIRs used to pass any number of command/argument pairings to the Device Driver. Command tables are terminated with the ADI_DEV_CMD_END macro.

The Application uses command tables to send specific, predefined MAX1233 Device Driver commands to configure the ADSP-BF527 processor, the ADSP-BF527 EZ-Kit Lite hardware and the MAX1233 IC to set various resources such as: device number, chip select, baud rate, interrupt flag pin, interrupt channel assignment, etc.

8.2.6.2 ADI_DEV_CMD_REGISTER_TABLE_WRITE

A register table is an array of ADI_DEV_ACCESS_REGISTER register ID/value pairs used to access device registers. Register tables are terminated with the ADI_DEV_REGEND macro.

The Application uses register tables to write registers on the MAX1233 device to control various functions and modes of operation.

8.2.6.3 ADI_DEV_CMD_REGISTER_TABLE_READ

A register table is an array of ADI_DEV_ACCESS_REGISTER register ID/value pairs used to access device registers. Register tables are terminated with the ADI_DEV_REGEND macro.

The Application uses register tables to read registers on the MAX1233 device to obtain conversion data and device status information.

8.2.7 Use the MAX1233 Device Driver

Once the MAX1233 is configured, the Application code typically enters a polling mode to await some exit condition such as a exit pushbutton press.

While polling to exit, the Application operation becomes event-driven and Application Callbacks are dispatched by the MAX1233 Device Driver in response to device events. The bulk of the Application work is done in the Application Callback.

8.2.8 Close the MAX1233 Device Driver

Once the exit condition is obtained, the Application should call adi_dev_Close() to release the MAX1233 Device Driver resources.

8.2.9 Terminate the System Services Library

Once the Application has called adi_dev_Close() to close the MAX1233 Device Driver, the System Services Library should also be released with a call to adi_ssl_Terminate().

8.3 Configuring the MAX1233 Device Driver on the ADSP-BF527 EZ-Kit Lite Evaluation Board

This code fragment is taken from the complete code Example file included with the MAX1233 Device Driver. The complete Example (including sources, include and project files) is located under the install tree, at:

"<InstallRoot>Analog Devices\VisualDSP 5.0\Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\touchscreen-keypad".

Please refer to the complete example for missing details, as some initializations and definitions are left out here for illustrative purpose. The excerpt illustrates use of the ADI_DEV_CMD_TABLE hardware configuration call to set the MAX1233 device number, chip select, baud rate, and flag/interrupt settings.

```
/* Port Info macros for PENIRQ */
#define     PENIRQ_FLAG_ID              ADI_FLAG_PF10       /* PENIRQ signal connected to this Flag */
#define     PENIRQ_FLAG_INTERRUPT_ID    ADI_INT_PORTF_INTA  /* Interrupt ID for PENIRQ Flag         */

/* Port Info macros for KEYIRQ */
#define     KEYIRQ_FLAG_ID              ADI_FLAG_PF9        /* KEYIRQ signal connected to this Flag */
#define     KEYIRQ_FLAG_INTERRUPT_ID    ADI_INT_PORTF_INTB  /* Interrupt ID for KEYIRQ Flag         */

// Interrupt ports for flag and interrupt designations
MAX1233_INTERRUPT_PORT PenIrqPort;    /* structure to hold Port Info for PENIRQ */
MAX1233_INTERRUPT_PORT KeyIrqPort;    /* structure to hold Port Info for KEYIRQ */

/* Port Info for PENIRQ */
PenIrqPort.FlagId       = PENIRQ_FLAG_ID;
PenIrqPort.FlagIntId    = PENIRQ_FLAG_INTERRUPT_ID;

/* Port Info for KYEYIRQ */
KeyIrqPort.FlagId       = KEYIRQ_FLAG_ID;
KeyIrqPort.FlagIntId    = KEYIRQ_FLAG_INTERRUPT_ID;

/* Pin I/O settings on the BF527 processor to communicate with the MAX1233 */
ADI_DEV_CMD_VALUE_PAIR  MAX1233_BF527_HardwareConfiguration[]=
{
  // configure Kookaburra pins to talk to the Max
  { ADI_MAX1233_CMD_SET_SPI_DEVICE_NUMBER, (void *)MAX1233_SPI_DEV_NUMBER }, /* SPI device number         */
  { ADI_MAX1233_CMD_SET_SPI_CS,            (void *)MAX1233_SPI_CS         }, /* SPI CS for MAX1233        */
  { ADI_MAX1233_CMD_SET_SPI_BAUD,          (void *)&SpiBaudRateRegValue   }, /* override default SPI baud */
  { ADI_MAX1233_CMD_INSTALL_PENIRQ,        (void *)&PenIrqPort            }, /* Enable PENIRQ monitoring  */
  { ADI_MAX1233_CMD_INSTALL_KEYIRQ,        (void *)&KeyIrqPort            }, /* Enable KEYIRQ Monitoring  */
  { ADI_DEV_CMD_END,                               NULL                   }  /* Terminate table           */
};

if (ADI_DEV_RESULT_SUCCESS !=
  (Result = adi_dev_Control(MAX1233DriverHandle,
                            ADI_DEV_CMD_TABLE,
                           (void *)MAX1233_BF527_HardwareConfiguration))) {
    printf("Failed to configure MAX1233 driver: 0x%08X\n",Result);
    exit(Result);
  }
```

9 Programming Example

This section provides some further Application Level programming examples to illustrate use of the MAX1233 Device Driver. The MAX1233 is assumed to already have been opened and configured for operation. The Application is also assumed to have entered a polling loop, awaiting some exit condition.

The example described here is the core of the Application: the Application Callback. The Application Callback is registered as part of the initial Device Open call to adi_dev_Open(). Having configured flags and interrupts, the MAX1233 dispatches Application Callbacks as PEN and KEY interrupts are received. It is the job of the Application Callback to process these events.

Please refer to the MAX1233 Device Driver include file for definitions of various macros for register name, bit values, and bit positions used in the example:

```
#include <drivers/touchscreen/adi_max1233.h>  /* MAX1233 driver includes */
```

9.1 ApplicationCallback Parameters

The ApplicationCallback has the following parameters:

9.1.1 Void *AppHandle

This is the Device Handle used in the adi_dev_Open() call of this device. The Device Handle is passed back to the Application to identify which device issued the callback, in the case that many devices share a common Application Callback. In this example, there is only one open device and so this parameter is safely ignored.

9.1.2 u32 Event

This is the event ID that caused the Callback to occur. In the case of the MAX1233 Device Driver, there are only two possible event IDs issued:

ADI_MAX1233_EVENT_PENIRQ_NOTIFICATION, and
ADI_MAX1233_EVENT_KEYIRQ_NOTIFICATION.

It is the responsibility of the Application Callback to distinguish action based on the event ID passed to the Application Callback. Typically, the Application Callback will employ a switch statement to take event-specific actions.

9.1.3 Void *pArg

The argument is an event-specific parameter which is defined by the Device Driver. In the case of the MAX1233 Device Driver example, the argument parameter is not used, as the event ID is sufficient.

9.2 Processing the Event

Although the Example illustrates event processing for both PEN and KEY Callback Events, only the PEN event processing is described here, as the KEY processing closely mirrors that of the PEN.

In the case of the PENIRQ (ADI_MAX1233_EVENT_PENIRQ_NOTIFICATION event), the Application Callback must respond by obtaining the pen location, as a minimum. Other actions may be performed to obtain pen pressure, battery voltage, etc., but only pen location is described here.

On detection of a pen event, the Application Callback enters a loop to track possible pen drag and drop activety. This is illustrated by the following code fragment:

```
static void ApplicationCallback( void *AppHandle, u32 Event, void *pArg){
  u32 Result;
  int busy;
  int hold;

  /* CASEOF (event type) */
  switch (Event) {
    case ADI_MAX1233_EVENT_PENIRQ_NOTIFICATION:
      hold=0;
        while (1) {  // track drag & drop activity
           …bulk of example appears here…
        }
    …
    break;
    …
  }  // end switch
…
```

Eventually, this loop is broken out by monitoring the pen status and detecting that the pen has been lifted.

9.3 Sync Output

Upon entry to the “while (pen down)” loop, the example optionally drives an external sync pulse to fire an oscilloscope or logic analyzer. This is a helpful diagnostic aid in debugging SPI transactions or monitoring noise on the touch screen force and sense lines. Of course, the SYNCBIT macro would be disabled for “production” release.

```
#ifdef SYNCBIT
  /* Pulse the sync bit for triggering the logic analyzer */
  if((Result = adi_flag_Clear(SYNC_BIT)) != ADI_FLAG_RESULT_SUCCESS) {
    printf("Failed to clear Flag pin connected to Sync Bit, Error Code: 0x%08X\n",Result);
    exit(Result);
  }
  if((Result = adi_flag_Set(SYNC_BIT)) != ADI_FLAG_RESULT_SUCCESS) {
    printf("Failed to set Flag pin connected to Sync Bit, Error Code: 0x%08X\n",Result);
    exit(Result);
  }
#endif
…
```

9.4 Test A Status Bit

This is a generic support function to test if a particular status bit is set in a particular status register. This routine is used in the following sections. The state of a single bit within a single register on the MAX1233 IC is returned.

```
static bool TestStatusBit(u16 StatusRegisterAddress, u16 StatusBit)
{
  // Return bool indicating state of status bit in register
  u32 Result;
  ADI_DEV_ACCESS_REGISTER Readback;

  Readback.Address = StatusRegisterAddress;
  Readback.Data = 0;

  // read ADSTS bits and return boolean if scan is complete
  if (ADI_DEV_RESULT_SUCCESS !=
    (Result = adi_dev_Control(MAX1233DriverHandle, ADI_DEV_CMD_REGISTER_READ, (void *)&Readback))) {
      printf("Failed to read MAX1233 status: 0x%08X\n",Result);
  }

  return ((Readback.Data & StatusBit) ? true : false);
}
```

9.5 Run ADC

The MAX1233 IC has two primary data conversion modes, controlled by the PENSTS and ADSTS bits of the ADC Control Register (at address 0x0040). Mode1 instructs the controller to wait for a pen touch and issues an interrupt (no conversion is made). Mode0 initiates a manual scan. There is a Mode2, which combines Modes0 and Mode1 (interrupt *and* scan), but this is not illustrated in the example. (Mode3 is a power-down standby mode). Note that these Mode bits have different read/write behaivor.

This example uses Mode1 as the primary state, forcing the MAX IC to simply issue an interrupt without any conversion. As the Application Callback receives the pen interrupt, it initiates a manual scan within the while loop by moving the controller into Mode0 and back again.

```
ADI_DEV_ACCESS_REGISTER MAX1233_InitiatePenConversion[] = {
  { MAX1233_REG_ADC, DEFAULT_ADC_BITS | (MAX1233_VAL_ADSTS0 << MAX1233_POS_ADSTS) },
  { ADI_DEV_REGEND, 0 }
};

// Run the ADC manually so we gate exactly one conversion at a time,
// allowing the interrupt to reset.  The interrupt is self-clearing
// on the Maxim after the conversion is complete.

// make sure converter is available
for (busy=0; false == TestStatusBit(MAX1233_REG_ADC, MAX1233_ADSTS_BIT); busy++);

// initiate a manual scan
if (ADI_DEV_RESULT_SUCCESS !=
  (Result = adi_dev_Control(MAX1233DriverHandle,
                            ADI_DEV_CMD_REGISTER_TABLE_WRITE,
                            (void *)MAX1233_InitiatePenConversion))) {
    printf("Failed to start MAX1233 scan sequence: 0x%08X\n",Result);
  }

// wait for scan to complete
for (busy = 0; false == TestStatusBit(MAX1233_REG_ADC, MAX1233_ADSTS_BIT); busy++);
...
               .
```

9.6 Read Conversion Data

Once the ADC busy bit clears, we proceed to read the conversion results and (in this case, simply) print them out.

```
ADI_DEV_ACCESS_REGISTER MAX1233_ReadPenData[] = {
  { MAX1233_REG_X,  0 },
  { MAX1233_REG_Y,  0 },
  { ADI_DEV_REGEND, 0 }
};

// read the conversion results; clearing the unhandled pending interrupt from the manual conversion
if( ADI_DEV_RESULT_SUCCESS !=
  (Result = adi_dev_Control(MAX1233DriverHandle,
                            ADI_DEV_CMD_REGISTER_TABLE_READ,
                            (void *)MAX1233_ReadPenData))) {
    printf("Failed to read MAX1233 device: 0x%08X\n",Result);
  }

// print results
if (hold ) {
  printf("Hold/drag at (x, y) = (%03d, %03d)\n", MAX1233_ReadPenData[0].Data, MAX1233_ReadPenData[1].Data);
} else {
  printf("Pen TOUCH at (x, y) = (%03d, %03d)\n", MAX1233_ReadPenData[0].Data, MAX1233_ReadPenData[1].Data);
}
...
```

9.7 Return ADC to Mode1

Place the MAX1233 IC back into the Mode1 state, in which it simply waits for further pen events.

```
ADI_DEV_ACCESS_REGISTER MAX1233_DetectPenTouchAndInterrupt[] = {
  { MAX1233_REG_ADC, (DEFAULT_ADC_BITS | (MAX1233_VAL_ADSTS1 << MAX1233_POS_ADSTS)) },
  { ADI_DEV_REGEND, 0 }
};

// return ADC to detect/interrupt mode (mode1) to unblock
// pen status bits needed to test if pen is still down
if (ADI_DEV_RESULT_SUCCESS !=
  (Result = adi_dev_Control(MAX1233DriverHandle,
                            ADI_DEV_CMD_REGISTER_TABLE_WRITE,
                            (void *)MAX1233_DetectPenTouchAndInterrupt))) {
  printf("Failed to put MAX1233 back into interrupt mode: 0x%08X\n",Result);
}
…
```

9.8 Test Pen Hold Status

Check if the pen is still down and update the “hold” variable to select print format for pen position. If the pen status is clear, break out of the loop.

```
// test pen status
if (TestStatusBit(MAX1233_REG_ADC, MAX1233_PENSTS_BIT))
  hold++;              // pen still down, continue tracking
else
  break;               // break out on pen-up status
...
```

9.9 Dummy Read

Do a “dummy read” to clear any blocked interrupts.

```
ADI_DEV_ACCESS_REGISTER MAX1233_DummyRead[] = {
  { MAX1233_REG_X,   0 },
  { MAX1233_REG_KPD, 0 },
  { ADI_DEV_REGEND,  0 }
};

/**************** Do a dummy read to unblock any pending MAX1233 interrupts ***************/
if (ADI_DEV_RESULT_SUCCESS  !=
  (Result = adi_dev_Control(MAX1233DriverHandle,
                            ADI_DEV_CMD_REGISTER_TABLE_READ,
                            (void *)MAX1233_DummyRead))) {
    printf("Failed to read MAX1233 device: 0x%08X\n",Result);
  }
...
```

9.10 Re-Enable Interrupts

Re-enable the interrupt input on the ADSP-BF527.

```
ADI_DEV_CMD_VALUE_PAIR  MAX1233_ReenablePENIRQ[]= {
  { ADI_MAX1233_CMD_REENABLE_PENIRQ, (void *)&PenIrqPort },
  { ADI_DEV_CMD_END, NULL }
};

// re-enable interrupting on PENIRQ input (which was disabled in the driver callback)
if (ADI_DEV_RESULT_SUCCESS !=
  (Result = adi_dev_Control(MAX1233DriverHandle,
                            ADI_DEV_CMD_TABLE,
                            (void *)MAX1233_ReenablePENIRQ))) {
    printf("Failed to start MAX1233 device: 0x%08X\n",Result);
  }

  break;  // end case (ADI_MAX1233_EVENT_PENIRQ_NOTIFICATION)
…
```