Actel SmartFusion™ MSS ACE Driver User's Guide

Version 2.1

Actel
POWER MATTERS

Actel Corporation, Mountain View, CA 94043

Printed in the United States of America

Part Number: 50200196-2

Release: June 2010

Trademarks

Table of Contents

Introduction

The SmartFusion™ microcontroller subsystem (MSS) includes the analog compute engine (ACE) which provides access to the analog capabilities of SmartFusion from the ARM® Cortex™-M3 microcontroller. This driver provides a set of functions for controlling the MSS ACE as part of a bare metal system where no operating system is available. These drivers can be adapted for use as part of an operating system, but the implementation of the adaptation layer between this driver and the operating system's driver model is outside the scope of this driver. The ACE includes:

- A sample sequencing engine (SSE) controlling the operations of up to three analog-to-digital converters (ADC)
- A post processing engine (PPE) processing the analog inputs samples generated as a result of the SSE operations
- An interface for controlling sigma delta DACs (SDD)
- An interface for controlling high-speed comparators

The sample sequencing engine controls the sampling of the various analog inputs based on a predefined sampling sequence without requiring intervention from the Cortex-M3. The sampling sequence is defined using the ACE configurator provided as part of the MSS Configurator software tool.

Available analog inputs are:

- Active bipolar prescaler inputs (ABPS) allowing to measure voltages within four possible ranges:
    - -15.36V to +15.36V (recommended input range is -11V to +14V)
    - -10.24V to +10.24V
    - -5.12V to +5.12V
    - -2.56V to +2.56V
- Current inputs
- Temperature inputs
- Direct ADC inputs allowing to measure a voltage between zero volts and the ADC's reference voltage (VAREF)

Refer to the analog front end section of the *SmartFusion Programmable Analog User's Guide* for further details about analog inputs.

The post processing engine can perform the following operations on the analog input samples generated as a result of the SSE operations:

- Calibration adjustment
- Averaging
- Threshold detection
- DMA transfer of most recent sample result to eSRAM or FPGA fabric

The result of analog input sampling is read from the PPE rather than directly from the ADC. This ensures more accurate sample results through the factory calibration adjustment performed by the PPE.

The PPE can be set to generate interrupts when specific threshold values are reached on analog inputs through the ACE configurator software tool. These thresholds can also be dynamically adjusted through the ACE driver.

The ACE provides an interface to the sigma delta DACs included within the AFE. This interface allows control of the DAC's output value. Dynamic configuration of the DAC is also possible.

The ACE provides an interface to the high speed comparators included within the AFE. This interface allows dynamic configuration of the comparators and controlling interrupts based on the comparators' state.

Features

The MSS ACE driver provides the following features:

- Reading analog input sampling results from the ACE's post processing engine
- Associating handler functions with post processing engine generated flags
- Controlling the SDD
- Associating interrupts with the state of the high- speed analog comparators
- Converting analog input sample value to and from real world units
- Controlling the ACE's SSE
- Controlling sample sequencing engine interrupts

The MSS ACE driver is provided as C-language source code.

Supported Hardware IP

The MSS ACE bare metal driver can be used with Actel's MSS_ACE IP version 0.8.7 or higher included in the SmartFusion MSS.

Files Provided

The files provided as part of the MSS ACE driver fall into three main categories: documentation, driver source code, and example projects. The driver is distributed via the Actel Firmware Catalog, which provides access to the documentation for the driver, generates the driver's source files into an application project, and generates example projects that illustrate how to use the driver.

Documentation

The Actel Firmware Catalog provides access to these documents for the driver:

- User's guide (this document)
- A copy of the end user license agreement for the driver source code
- Release notes

Driver Source Code

The Actel Firmware Catalog generates the driver's source code into a *drivers\mss_ace* subdirectory of the selected software project directory. The files making up the driver are detailed below.

mss_ace.h

This header file contains the public application programming interface (API) of the MSS ACE software driver. This file should be included in any C source file that uses the MSS ACE software driver.

mss_ace.c

This C source file contains the implementation of the MSS ACE software driver.

ace_flags.c

This C source file contains the implementation of the MSS ACE driver post processing engine flag control functions.

ace_sse.c

This C source file contains the implementation of the MSS ACE driver sample sequencing engine control functions.

ace_convert.c

This C source file contains the implementation of the MSS ACE driver conversion functions.

mss_ace_configurator.h

This header file contains the definitions of data structures used by the ACE configurator software tool for sharing information about the ACE configuration with the ACE driver. This header file does not need to be included in the application source code. It is only used within the ACE driver implementation.

mtd_data.h

This header file contains the definitions of data structures used to store factory calibration data into SmartFusion eNVM spare pages. This header file does not need to be included in the application source code. It is only used within the ACE driver implementation.

envm_layout.h

This header file contains the location of the factory calibration data. This header file does not need to be included in the application source code. It is only used within the ACE driver implementation.

ace_config.h

This header file contains definitions about the ACE configuration selected as part of the ACE configurator software tool. The ACE configurator is part of the SmartFusion MSS Configurator provided as part of the Libero® Integrated Design Environment (IDE) tools suite.

This file is generated by the ACE configurator to reflect the ACE configuration that was entered into the ACE configurator. This header file does not need to be included in the application source code. It is only used within the ACE driver implementation.

You may need to copy this file from the Libero IDE project's *firmware\drivers_config\mss_ace* directory into the *drivers_config\mss_ace* subdirectory of your software project if you are running the MSS Configurator software tool from Libero IDE.

ace_config.c

This C source file contains the configuration of the ACE selected in the ACE Configurator software tool. It contains data structures that are used by the ACE driver implementation to interface with the configured ACE hardware.

This file is generated by the ACE configurator to reflect the ACE configuration that was entered into the ACE configurator. This header file does not need to be included in the application source code. It is only used within the ACE driver implementation.

You will need to copy this file from the Libero project's *firmware\drivers_config\mss_ace* directory into the *drivers_config\mss_ace* subdirectory of your software project if you are running the MSS Configurator software tool from Libero IDE.

ace_handles.h

This header file contains the list of handles for the analog input channels and post processing flags. This header file does not need to be included in the application source code. It is only used within the ACE driver implementation. View this file's content to find out the name of the various channels and flags handlers.

This file is generated by the ACE configurator to reflect the ACE configuration that was entered into the ACE configurator. This header file does not need to be included in the application source code. It is only used within the ACE driver implementation.

You will need to copy this file from the Libero project's *firmware\drivers_config\mss_ace* directory into the *drivers_config\mss_ace* subdirectory of your software project if you are running the MSS Configurator software tool from Libero IDE.

Example Code

The Actel Firmware Catalog provides access to example projects illustrating the use of the driver. Each example project is self contained and is targeted at a specific processor and software toolchain combination. The example projects are targeted at the FPGA designs in the hardware development tutorials supplied with Actel's development boards. The tutorial designs can be found on the Actel Development Kit web page (www.actel.com/products/hardware).

Driver Deployment

This driver is intended to be deployed from the Actel Firmware Catalog into a software project by generating the driver's source files into the project directory. The driver uses the SmartFusion Cortex Microcontroller Software Interface Standard – Peripheral Access Layer (CMSIS-PAL) to access MSS hardware registers. You must ensure that the SmartFusion CMSIS-PAL is either included in the software tool chain used to build your project or is included in your project. The most up-to-date SmartFusion CMSIS-PAL files can be obtained using the Actel Firmware Catalog.

The following example shows the intended directory structure for a SoftConsole ARM Cortex-M3 project targeted at the SmartFusion MSS. This project uses the MSS ACE and MSS Watchdog drivers. Both of these drivers rely on SmartFusion CMSIS-PAL for accessing the hardware. The contents of the *drivers* directory result from generating the source files for each driver into the project. The contents of the *CMSIS* directory result from generating the source files for the SmartFusion CMSIS-PAL into the project. The contents of the *drivers_config* directory result from copying the ACE configurator generated files into the project.

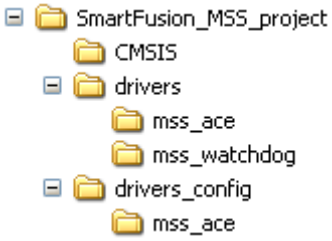

Figure 1 · SmartFusion MSS Project Example

Driver Configuration

The ACE driver requires configuration files generated by the ACE configurator. The ACE configurator is part of the SmartFusion MSS Configurator provided with the Libero IDE tools suite. The ACE Configurator is a user friendly graphical interface tool allowing configuring the ACE hardware. It generates the ACE and AFE configuration. The ACE configurator generates configuration data stored in SmartFusion eNVM and a set of C files. The eNVM stored configuration data is used by the SmartFusion system boot code to configure the ACE hardware on reset before giving control to the application. The C-language files generated by the ACE configurator must be copied into the software project to allow the ACE driver to interact with the configured ACE hardware.

The base address, register addresses and interrupt number assignment for the MSS ACE are defined as constants in the SmartFusion CMSIS-PAL. You must ensure that the SmartFusion CMSIS-PAL is either included in the software toolchain used to build your project or is included in your project.

Application Programming Interface

This section describes the driver's API. The functions and related data structures described in this section are used by the application programmer to control the MSS ACE peripheral from the user's application.

Theory of Operation

The configuration of the ACE is set though the use of the ACE configurator included in the SmartFusion MSS Configurator software tool provided as part of the Libero IDE tool suite. The ACE configurator offers an easy to use graphical method of selecting the configuration of the following ACE characteristics:

- Analog input channels configuration
- ADC configuration
- Analog input channels sampling sequence
- Filtering applied to analog input samples
- Threshold flags configuration including hysteresis or state filtering properties
- Selection of post processing results transferred though DMA
- Sigma delta DACs configuration
- Analog comparators configuration

The selected configuration hardware settings, SSE microcode, and PPE microcode are stored in the SmartFusion eNVM. This configuration data is used by the system boot to configure the ACE after the system come out of reset and before control is passed to the application. This results in the ACE being fully operational by the time the application starts executing.

The ACE configurator also generates a set of C files containing information about the ACE's configuration. These C files must be copied into the *drivers_config/mss_ace* folder of you r software project for consumption by the ACE driver. The ACE driver uses the content of these configuration files to interact with the configured ACE hardware.

The ACE driver functions are grouped into the following categories:

- Initialization
- Reading analog input channels values and properties
- PPE flags
- Conversion functions between sample value and real world units
- SSE control
- SSE interrupts Control
- Comparators control
- Sigma delta digital to analog converters control
- Direct analog block configuration and usage

Initialization

The ACE driver is initialized through a call to the *ACE_init()* function. The *ACE_init()* function must be called before any other ACE driver functions can be called. It initializes the ACE's internal data.

Reading Analog Input Channels Values and Properties

The ACE driver allows retrieving the most recent post processed sample value for each analog input. It also allows retrieving the name of the analog input channel assigned in the ACE Configurator and whether the input channel samples a voltage, current or temperature.

Each individual analog input channel is identified using a channel handle which is passed as parameter to the ACE input channel driver functions. The channel handles are design specific. The list of channel handles is generated by the ACE configurator based on the names given to the input signals. The channel handles can be found in the *drivers_config\mss_ace\ace_handles.h* file. The channel handle can be obtained from the channel name using the *ACE_get_channel_handle()* function. It is also possible to iterate through all the channels using the *ACE_get_channel_count()*, *ACE_get_first_channel()* and *ACE_get_next_channel()* functions.

Reading analog input samples from the post processing engine is done the following function:

- *uint16_t ACE_get_ppe_sample(ace_channel_handle_t channel_handle)*

Information about an input channel can be retrieved using the following functions:

- *const uint8_t * ACE_get_channel_name(ace_channel_handle_t channel_handle)*
- *channel_type_t ACE_get_channel_type(ace_channel_handle_t channel_handle)*

Post Processing Engine Flags

The SmartFusion ACE PPE provides the ability to monitor the state of analog input channels and detect when certain threshold values are crossed. Flags are raised by the PPE when these thresholds are crossed. Interrupts can optionally be generated when flags are raised.

The flags are defined using the ACE configurator software tool. The flag's name, threshold value and hysteresis settings are specified in the ACE configurator. The ACE configurator generates microcode based on the selected configuration which is executed at system run time by the PPE. The PPE microcode is loaded into the ACE at chip boot time by the Actel provided system boot code. No ACE driver intervention is required to load up the PPE microcode.

The ACE driver allows:

- Retrieving the current state of the post processing engine flags
- Assigning a handler function to individual flag assertions
- Assigning a handler function to flags generated based on the value of a specific channel
- Controlling flag interrupts
- Dynamically modify a flag's threshold value
- Dynamically modify a flag's hysteresis

Each individual flag is identified using a flag handle which is passed as parameter to the ACE driver functions controlling the flags. The flag handles are design specific. They are defined in the *drivers_config\mss_ace\ace_handles.h* file which is generated by the ACE configurator based on the names selected for the signal and flag names. A flag handle can be obtained from the driver using the name of the flag entered in the ACE Configurator software when the flag was created. A flag handle can also be obtained using the functions *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()* when iterating through the flags associated with an analog input channel. The functions available for retrieving flag handles are:

- *ace_flag_handle_t ACE_get_flag_handle (const uint8_t *p_sz_full_flag_name)*
- *uint32_t ACE_get_channel_flag_count (ace_channel_handle_t channel_handle)*
- *ace_flag_handle_t ACE_get_channel_first_flag (ace_channel_handle_t channel_handle, uint16_t *iterator)*
- *ace_flag_handle_t ACE_get_channel_next_flag (ace_channel_handle_t channel_handle, uint16_t *iterator)*

The current status of a flag can be polled using the following function:

- *int32_t ACE_get_flag_status (ace_flag_handle_t flag_handle)*

Interrupt handlers can be registered with the ACE driver to handle individual flags. These interrupt handlers will be called by the ACE driver when a specific flag is raised. The flag interrupt control functions are:

- *void ACE_register_flag_isr (ace_flag_handle_t flag_handle, flag_isr_t flag_isr)*
- *void ACE_enable_flag_irq (ace_flag_handle_t flag_handle)*
- *void ACE_disable_flag_irq (ace_flag_handle_t flag_handle)*
- *void ACE_clear_flag_irq (ace_flag_handle_t flag_handle)*

Interrupt handlers can be registered with the ACE driver to handle all flags associated with one specific analog input channel. These interrupt handlers will be called by the ACE driver when one of the flags, generated based on the state of the specified analog input channel, is raised. The channel flag interrupt control functions are:

- *void ACE_register_channel_flags_isr (ace_channel_handle_t channel_handle, channel_flag_isr_t channel_flag_isr)*
- *void ACE_enable_channel_flags_irq (ace_channel_handle_t channel_handle)*
- *void ACE_disable_channel_flags_irq (ace_channel_handle_t channel_handle)*
- *void ACE_clear_channel_flags_irq (ace_channel_handle_t channel_handle)*

A single global interrupt handler can be registered with the ACE driver. The global flag interrupt handler function will be called by the ACE driver when any of the interrupt enabled flag is raised. The handle of the flag causing the interrupt and the handle of the associated analog input channel is passed as parameter to the registered global flag handler.

- *void ACE_register_global_flags_isr (global_flag_isr_t global_flag_isr)*

The configuration of a flag can be dynamically modified using the following functions:

- *void ACE_set_flag_threshold (ace_flag_handle_t flag_handle, uint16_t new_threshold)*
- *void ACE_set_flag_hysteresis (ace_flag_handle_t flag_handle, uint16_t adc_hysteresis)*
- *void ACE_set_channel_hysteresis (ace_channel_handle_t channel_handle, uint16_t adc_hysteresis)*
- *void ACE_set_flag_assertion(ace_flag_handle_t flag_handle, uint16_t assertion_value)*
- *void ACE_set_flag_deassertion(ace_flag_handle_t flag_handle, uint16_t assertion_value)*

Information about a flag can be retrieved using the following functions once the flag handle is known:

- *const uint8_t * ACE_get_flag_name (ace_flag_handle_t flag_handle)*
- *ace_channel_handle_t ACE_get_flag_channel (ace_flag_handle_t flag_handle)*
- *uint32_t ACE_is_hysteresis_flag (ace_flag_handle_t flag_handle)*
- *uint32_t ACE_is_under_flag (ace_flag_handle_t flag_handle)*

Conversion To and From Real World Units

The ACE driver provides a set of conversion functions to convert sample values read from the post processing engine into real world units:

- millivolts
- milliamps
- Degrees Kelvin
- Degrees Celsius
- Degrees Fahrenheit

Conversion functions are also available to convert from real world units into PPE sample values. These functions are typically used for dynamically adjusting flags threshold values.

Sample Sequencing Engine Control

The ACE driver provides a set of functions for dynamically controlling the SSE. These functions are only required for managing multiple sampling sequences. The use of these functions is not required for most applications since the SSE is already configured and running by the time the application starts.

Sample Sequencing Engine Interrupts Control

The ACE driver provides a set of functions for managing interrupts generated by the SSE. These functions allow enabling, disabling and clearing interrupt defined as part of the sampling sequence. These functions also allow controlling interrupts generated by the ADCs.

Comparators Control

The ACE driver provides a set of functions for managing interrupts generated based on the change of state of the high speed comparators. Functions are also provided to dynamically modify the comparators configuration.

Sigma Delta Digital to Analog Converters Control

The ACE driver provides functions for controlling the output value of the SDD. Functions are also provided for dynamically adjusting the configuration of the SDDs.

Types

adc_channel_id_t

Prototype

```
typedef enum {
  ADC0_1P5V = 0,        /*!< Analog Module 0, 1.5V/GND */
  ABPS0 = 1,            /*!< Analog Module 0, Quad0 Active Bipolar Pre-Scaler input 1 */
  ABPS1 = 2,            /*!< Analog Module 0, Quad0 Active Bipolar Pre-Scaler input 2 */
  CM0 = 3,              /*!< Analog Module 0, Quad0 Current Monitor Block */
  TM0 = 4,              /*!< Analog Module 0, Quad0 Temperature Monitor Block */
  ABPS2 = 5,            /*!< Analog Module 0, Quad1 Active Bipolar Pre-Scaler input 1 */
  ABPS3 = 6,            /*!< Analog Module 0, Quad1 Active Bipolar Pre-Scaler input 2 */
  CM1 = 7,              /*!< Analog Module 0, Quad1 Current Monitor Block */
  TM1 = 8,              /*!< Analog Module 0, Quad1 Temperature Monitor Block */
  ADC0 = 9,             /*!< Analog Module 0 Direct Input 0 */
  ADC1 = 10,            /*!< Analog Module 0 Direct Input 1 */
  ADC2 = 11,            /*!< Analog Module 0 Direct Input 2 */
  ADC3 = 12,            /*!< Analog Module 0 Direct Input 3 */
  SDD0_IN = 15,         /*!< Analog Module 0 Sigma-Delta DAC output */

  ADC1_1P5V = 16,       /*!< Analog Module 1, 1.5V/GND */
  ABPS4 = 17,           /*!< Analog Module 1, Quad0 Active Bipolar Pre-Scaler input 1 */
  ABPS5 = 18,           /*!< Analog Module 1, Quad0 Active Bipolar Pre-Scaler input 2 */
  CM2 = 19,             /*!< Analog Module 1, Quad0 Current Monitor Block */
  TM2 = 20,             /*!< Analog Module 1, Quad0 Temperature Monitor Block */
  ABPS6 = 21,           /*!< Analog Module 1, Quad1 Active Bipolar Pre-Scaler input 1 */
  ABPS7 = 22,           /*!< Analog Module 1, Quad1 Active Bipolar Pre-Scaler input 2 */
  CM3 = 23,             /*!< Analog Module 1, Quad1 Current Monitor Block */
  TM3 = 24,             /*!< Analog Module 1, Quad1 Temperature Monitor Block */
  ADC4 = 25,            /*!< Analog Module 1 Direct Input 0 */
  ADC5 = 26,            /*!< Analog Module 1 Direct Input 1 */
  ADC6 = 27,            /*!< Analog Module 1 Direct Input 2 */
  ADC7 = 28,            /*!< Analog Module 1 Direct Input 3 */
  SDD1_IN = 31,         /*!< Analog Module 1 Sigma-Delta DAC output */

  ADC2_1P5V = 32,       /*!< Analog Module 2, 1.5V/GND */
  ABPS8 = 33,           /*!< Analog Module 2, Quad0 Active Bipolar Pre-Scaler input 1 */
  ABPS9 = 34,           /*!< Analog Module 2, Quad0 Active Bipolar Pre-Scaler input 2 */
  CM4 = 35,             /*!< Analog Module 2, Quad0 Current Monitor Block */
  TM4 = 36,             /*!< Analog Module 2, Quad0 Temperature Monitor Block */
  ABPS10 = 37,          /*!< Analog Module 2, Quad1 Active Bipolar Pre-Scaler input 1 */
  ABPS11 = 38,          /*!< Analog Module 2, Quad1 Active Bipolar Pre-Scaler input 2 */
  CM5 = 39,             /*!< Analog Module 2, Quad1 Current Monitor Block */
  TM5 = 40,             /*!< Analog Module 2, Quad1 Temperature Monitor Block */
```

```
  ADC8 = 41,            /*!< Analog Module 2 Direct Input 0 */
  ADC9 = 42,            /*!< Analog Module 2 Direct Input 1 */
  ADC10 = 43,           /*!< Analog Module 2 Direct Input 2 */
  ADC11 = 44,           /*!< Analog Module 2 Direct Input 3 */
  SDD2_IN = 47,         /*!< Analog Module 2 Sigma-Delta DAC output */
  INVALID_CHANNEL = 255 /*!< Used to indicate errors */
} adc_channel_id_t;
```

Description

ADC channel IDs. This enumeration is used to identify the ADC's analog inputs. It caters for up to three ADCs/analog modules as can be found on the larger parts of the SmartFusion family. The channel ID numbering is designed to allow easy extraction of the ADC number and also the individual ADC input number by simple shifting and masking. This enumeration is used as parameter to the *ACE_get_input_channel_handle()* function retrieving the channel handle associated with a specific analog input signal.

ace_channel_handle_t

Prototype

```
typedef enum {
  <USER_DEFINED_CHANNEL_NAMES>,
  NB_OF_ACE_CHANNEL_HANDLES = 0
} ace_channel_handle_t;
```

Description

The members of this enumeration provide the ACE driver with handles identifying the input channels configured using the ACE configurator. The channel handles are an enumerated representation of the names selected for the analog input signals during configuration with the ACE configurator. Channel handles are used as parameter to ACE driver functions dealing with analog input channels to identify the analog input channel the function should use.

For example, adding signals named “MainPowerSupply”, “BatteryBackup” and “AmbientTemperature” to the ACE configuration in the ACE configurator will result in the following enumeration being generated by the ACE configurator:

```
typedef enum {
    MainPowerSupply = 0,
    BatteryBackup,
    AmbientTemperature,
    NB_OF_ACE_CHANNEL_HANDLES
} ace_channel_handle_t;
```

These enumeration members can then be used as parameter to channel handling functions such as the *ACE_get_ppe_sample()* function to read sample values as follows:

```
uint16_t supply;
uint16_t battery;
uint16_t temperature;
supply = ACE_get_ppe_sample( MainPowerSupply );
battery  = ACE_get_ppe_sample( BatteryBackup );
temperature = ACE_get_ppe_sample( AmbientTemperature );
```

The NB_OF_ACE_CHANNEL_HANDLES member is always present in the enumeration and indicates the number of ADC input channels that are currently configured.

Note: The ACE configurator generates the ace_handles.h file into the *.\drivers_config\mss_ace* folder of the firmware project. This file contains the custom the *ace_channel_handle_t* enumeration for your ACE configuration. The ACE driver automatically includes this file when the *.\drivers_config\mss_ace* folder is present in the firmware project.

ace_flag_handle_t

Prototype

```
typedef enum {
  <USER_DEFINED_FLAG_NAMES>,
  NB_OF_ACE_FLAG_HANDLES = 0
} ace_flag_handle_t;
```

Description

The members of this enumeration provide the ACE driver with handles identifying the threshold flags configured using the ACE configurator. The flag handles are the concatenated input signal and threshold flag names entered when configuring the threshold flags.

For example, following from the *ace_channel_handle_t* example above, configuring threshold flags named "FullyCharged" and "Low" for channel the named "BatteryBackup" and configuring threshold flags "Hot", "Warm" and "Cool" for channel "AmbientTemperature" will result in the following ace_flag_handle_t enumeration being generated by the ACE configurator:

```
typedef enum {
    BatteryBackup_FullyCharged,
    BatteryBackup_Low,
    AmbientTemperature_Hot,
    AmbientTemperature_Warm,
    AmbientTemperature_Cool,
    NB_OF_ACE_FLAG_HANDLES = 0
} ace_flag_handle_t;
```

These enumeration members can be used with flag handling functions such as *ACE_get_flag_status()* as follows:

```
if ( FLAG_ASSERTED == ACE_get_flag_status( AmbientTemperature_Hot )
{
      switch_fan_on();
}
if ( FLAG_ASSERTED == ACE_get_flag_status( AmbientTemperature_Warm )
{
      switch_fan_off();
      switch_heater_off();
}
if( FLAG_ASSERTED == ACE_get_flag_status( AmbientTemperature_Cool )
{
      switch_heater_on();
}
```

The NB_OF_ACE_FLAG_HANDLES member is always present in the enumeration and indicates the number of threshold flags that are currently configured.

Note: The ACE configurator generates the ace_handles.h file into the *.\drivers_config\mss_ace* folder of the firmware project. This file contains the custom the *ace_flag_handle_t* enumeration for your ACE configuration. The ACE driver automatically includes this file when the *.\drivers_config\mss_ace* folder is present in the firmware project.

channel_type_t

Prototype

```
typedef enum {
  VOLTAGE,
  CURRENT,
  TEMPERATURE
} channel_type_t;
```

Description

The *channel_type_t* enumeration is used to identify the type of quantity measured by an analog input channel. It is typically used to figure out the type of conversion that must be applied to the ADC value generated from sampling a channel in order to yield real world units such millivolts, milliamps or degrees.

flag_isr_t

Prototype

```
void (*flag_isr_t)( ace_flag_handle_t flag_handle );
```

Description

This defines the function prototype that must be followed by MSS ACE Post Processing Engine (PPE) flag handler functions. These functions are registered with the ACE driver and associated with a particular flag through the *ACE_register_flag_isr()* function. The ACE driver will call the flag handler function when the associated flag is raised.

Declaring and Implementing PPE Flag Handler Functions

PPE flag handler functions should follow the following prototype:

```
void my_flag_handler ( ace_flag_handle_t flag_handle );
```

The actual name of the PPE flag handler is unimportant. You can use any name of your choice for the PPE flag handler.

The *flag_handle* parameter passes the handle of the raised flag to the flag handler function.

channel_flag_isr_t

Prototype

```
void (*channel_flag_isr_t)( ace_flag_handle_t flag_handle )
```

Description

This defines the function prototype that must be followed by MSS ACE PPE channel flag handler functions. These functions are registered with the ACE driver and associated with a particular ADC input channel through the *ACE_register_channel_flags_isr()* function. The ACE driver will call the channel flags handler function when one of the flags for the associated ADC input channel is raised.

Declaring and Implementing PPE Channel Flag Handler Functions

PPE channel flag handler functions should follow the following prototype:

```
void my_channel_flag_handler ( ace_flag_handle_t flag_handle );
```

The actual name of the PPE channel flag handler is unimportant. You can use any name of your choice for the PPE channel flag handler. The *flag_handle* parameter passes the handle of the raised flag to the channel flag handler function.

global_flag_isr_t

Prototype

void(* global_flag_isr_t)(ace_flag_handle_t flag_handle, ace_channel_handle_t channel_handle)

Description

This defines the function prototype that must be followed by MSS ACE PPE global flag handler functions. These functions are registered with the ACE driver through the *ACE_register_global_flags_isr()* function. The ACE driver will call the global flags handler function when any flag for any ADC input channel is raised.

Declaring and Implementing Global Flag Handler Functions

PPE global flag handler functions should follow the following prototype:

```
void my_global_flag_handler(
  ace_flag_handle_t flag_handle,
  ace_channel_handle_t channel_handle
);
```

The actual name of the PPE global flag handler is unimportant. You can use any name of your choice for the PPE global flag handler. The *flag_handle* parameter passes the handle of the raised flag to the global flag handler function. The *channel_handle* parameter passes the handle of the channel for which the flag was raised to the global flag handler function.

sse_sequence_handle_t

Prototype

```
typedef uint16_t sse_sequence_handle_t
```

Description

The SSE control functions use a parameter of this type as a handle to identify the SSE sequences configured using the ACE configurator. The *ACE_get_sse_seq_handle()* function retrieves the handle of the SSE sequence identified by the sequence name passed as parameter.

Note: The ACE configurator generates ACE driver configuration files into the .*drivers_config**mss_ace* folder of the firmware project. These files contain the details of the SSE sequence handles for your ACE configuration. The ACE driver automatically includes these files when the .*drivers_config**mss_ace* folder is present in the firmware project.

sse_irq_id_t

Prototype

```
typedef enum {
  PC0_FLAG0 = 0,
  PC0_FLAG1 = 1,
  PC0_FLAG2 = 2,
  PC0_FLAG3 = 3,
  PC1_FLAG0 = 4,
  PC1_FLAG1 = 5,
  PC1_FLAG2 = 6,
  PC1_FLAG3 = 7,
  PC2_FLAG0 = 8,
  PC2_FLAG1 = 9,
  PC2_FLAG2 = 10,
  PC2_FLAG3 = 11,
  ADC0_DATAVALID = 12,
  ADC1_DATAVALID = 13,
  ADC2_DATAVALID = 14,
  ADC0_CALIBRATION_COMPLETE = 15,
  ADC1_CALIBRATION_COMPLETE = 16,
  ADC2_CALIBRATION_COMPLETE = 17,
  ADC0_CALIBRATION_START = 18,
  ADC1_CALIBRATION_START = 19,
  ADC2_CALIBRATION_START = 20,
  NB_OF_SSE_FLAG_IRQS = 21
} sse_irq_id_t;
```

Description

The *sse_irq_id_t* enumeration is used to identify the SSE interrupt sources to the SSE interrupt control functions.

comparator_id_t

Prototype

```
typedef enum {
  CMP0 = 0,                    /*!< Analog module 0, Quad 0, CMB comparator */
  CMP1 = 1,                    /*!< Analog module 0, Quad 0, TMB comparator */
  CMP2 = 2,                    /*!< Analog module 0, Quad 1, CMB comparator */
  CMP3 = 3,                    /*!< Analog module 0, Quad 1, TMB comparator */
  CMP4 = 4,                    /*!< Analog module 1, Quad 0, CMB comparator */
  CMP5 = 5,                    /*!< Analog module 1, Quad 0, TMB comparator */
  CMP6 = 6,                    /*!< Analog module 1, Quad 1, CMB comparator */
  CMP7 = 7,                    /*!< Analog module 1, Quad 1, TMB comparator */
  CMP8 = 8,                    /*!< Analog module 2, Quad 0, CMB comparator */
  CMP9 = 9,                    /*!< Analog module 2, Quad 0, TMB comparator */
  CMP10 = 10,                  /*!< Analog module 2, Quad 1, CMB comparator */
  CMP11 = 11,                  /*!< Analog module 2, Quad 1, TMB comparator */
  NB_OF_COMPARATORS = 12
} comparator_id_t;
```

Description

The *comparator_id_t* enumeration is used by the comparator control functions to identify the analog comparators included in the SmartFusion analog block.

comp_hysteresis_t

Prototype

```
typedef enum {
  NO_HYSTERESIS = 0,
  HYSTERESIS_10_MV = 1,
  HYSTERESIS_30_MV = 2,
  HYSTERESIS_100_MV = 3,
  NB_OF_HYSTERESIS = 4
} comp_hysteresis_t ;
```

Description

The *comp_hysteresis_t* enumeration is used by the *ACE_set_comp_hysteresis()* function to set the hysteresis of the analog comparators included in the SmartFusion analog block. This enumeration provides the allowed values of the *hysteresis* parameter of the *ACE_set_comp_hysteresis()* function.

comp_reference_t

Prototype

```
typedef enum
{
  SDD0_COMP_REF = 0,          /*!< Analog Module 0 Sigma Delta DAC output */
  SDD1_COMP_REF = 1,          /*!< Analog Module 1 Sigma Delta DAC output */
  SDD2_COMP_REF = 2,          /*!< Analog Module 2 Sigma Delta DAC output */
  ADC_IN_COMP_REF = 3,        /*!< Direct ADC input */
  NB_OF_COMP_REF = 4
} comp_reference_t;
```

Description

The *comp_reference_t* enumeration is used by the *ACE_set_comp_reference()* function to select the reference input of the odd numbered analog comparators included in the SmartFusion analog block. This enumeration provides the allowed values of the *reference* parameter of the *ACE_set_comp_reference ()* function.

sdd_id_t

Prototype

```
typedef enum {
  SDD0_OUT = 0,     /*!< Analog Module 0 Sigma Delta DAC */
  SDD1_OUT = 1,     /*!< Analog Module 1 Sigma Delta DAC */
  SDD2_OUT = 2,     /*!< Analog Module 2 Sigma Delta DAC */
  NB_OF_SDD = 3
} sdd_id_t;
```

Description

The *sdd_id_t* enumeration is used to identify the sigma delta DACs to the SDD control functions, *ACE_configure_sdd()*, *ACE_enable_sdd()*, *ACE_disable_sdd()* and *ACE_set_sdd_value()*. There is one SDD per analog module.

sdd_resolution_t

Prototype

```
typedef enum {
  SDD_8_BITS = 0,
  SDD_16_BITS = 4,
  SDD_24_BITS = 8
} sdd_resolution_t;
```

Description

The *sdd_resolution_t* enumeration is used as a parameter to the *ACE_configure_sdd()* function to specify DAC resolution of the sigma delta DAC.

sdd_update_method_t

Prototype

```
typedef enum {
  INDIVIDUAL_UPDATE = 0,
  SYNC_UPDATE = 1
} sdd_update_method_t;
```

Description

The *sdd_update_method_t* enumeration is used as a parameter to the *ACE_configure_sdd()* function to specify individual or synchronous updating of the sigma delta DACs.

Constant Values

ADC Input Channel Handle Invalid

This constant returned by the *ACE_get_flag_channel()*, *ACE_get_channel_handle()* and *ACE_get_input_channel_handle()* functions when the driver can't find a valid handle for the ADC input channel.

Constant	Description
INVALID_CHANNEL_HANDLE	Cannot find a valid ADC input channel handle

Table 1 · ADC Input Channel Handle Invalid

Post Processing Engine (PPE) Flag Handle Invalid

This constant is returned by the *ACE_get_flag_handle* function when the driver can't find a valid handle for the PPE flag.

Constant	Description
INVALID_FLAG_HANDLE	Cannot find a valid PPE flag handle

Table 2 · PPE Flag Handle Invalid

Post Processing Engine (PPE) Flag Status

These constant definitions are the return values of the *ACE_get_flag_status()* function. They specify the status of the PPE flag.

Constant	Description
FLAG_ASSERTED	Flag is raised/asserted
FLAG_NOT_ASSERTED	Flag is not asserted
UNKNOWN_FLAG	Flag name is not recognized by the driver

Table 3 · Post Processing Engine Flag Status

Sample Sequencing Engine (SSE) Sequence Handle Invalid

This constant is returned by the *ACE_get_sse_seq_handle()* function when the driver can't find a valid handle for the SSE sequence.

Constant	Description
INVALID_SSE_SEQ_HANDLE	Cannot find a valid SSE sequence handle

Table 4 · SSE Sequence Handle Invalid

Sigma Delta DAC (SDD) Configurable Operating Modes

These constant definitions are used as an argument to the *ACE_configure_sdd()* function to specify operating mode of the sigma delta DAC.

Constant	Description
SDD_CURRENT_MODE	Current output
SDD_VOLTAGE_MODE	Voltage output
SDD_RETURN_TO_ZERO	Return To Zero (RTZ) enabled
SDD_NON_RTZ	RTZ not enabled

Table 5 · Sigma Delta DAC Operating Mode

Sigma Delta DAC (SDD) Output Unmodified

This constant definition is used as an argument to the *ACE_set_sdd_value_sync()* function to specify that the output value of SDD0, or SDD1, or SDD2 should not be modified.

Constant	Description
SDD_NO_UPDATE	Do not change the SDD output

Table 6 · Sigma Delta DAC Output Unmodified

Data structures

There are no MSS ACE driver specific data structures

Functions − Initialization

ACE_init

Prototype

```
void ACE_init
(
  void
);
```

Description

The *ACE_init()* function initializes the SmartFusion MSS ACE driver. It initializes the ACE driver's internal data structures. The *ACE_init()* function must be called before any other MSS ACE driver functions can be called.

Parameters

This function takes no parameters.

Return Value

This function does not return a value.

Example

```
ACE_init();
```

Functions − Reading Analog Input Channels Values and Properties

The following functions are used to access analog input channels properties and sampled values.

- *ACE_get_channel_handle()*
- *ACE_get_input_channel_handle()*
- *ACE_get_channel_count()*
- *ACE_get_ppe_sample()*
- *ACE_get_channel_name()*
- *ACE_get_channel_type()*
- *ACE_get_first_channel()*
- *ACE_get_next_channel()*

ACE_get_channel_handle

Prototype

```
ace_channel_handle_t ACE_get_channel_handle
(
  const uint8_t * p_sz_channel_name
);
```

Description

The *ACE_get_channel_handle()* function returns the channel handle associated with an analog input channel name. The retrieved channel handle will be subsequently used as parameter to function *ACE_get_ppe_sample()* used to read the most recent post processed sample for the analog input identified through the channel/service name passed as argument to this function.

Parameters

p_sz_channel_name

The *p_sz_channel_name* parameter is a zero-terminated string containing the name of the channel/service as entered in the ACE configurator.

Return Value

This function returns a channel handle. This channel handle is required as parameter to function *ACE_get_ppe_sample()*. It will return INVALID_CHANNEL_HANDLE if the channel/service name is not recognized.

Example

```
uint16_t adc_result;
ace_channel_handle_t at0;
at0 = ACE_get_channel_handle("VoltageMonitorAT0");
adc_result = ACE_get_ppe_sample( at0 );
```

ACE_get_input_channel_handle

Prototype

```
ace_channel_handle_t ACE_get_input_channel_handle
(
  adc_channel_id_t channel_id
);
```

Description

The *ACE_get_input_channel_handle()* function returns the channel handle for the hardware analog input channel specified as parameter.

Parameters

channel_id

The *channel_id* parameter identifies a hardware analog input of the ACE.

Return Value

This function returns a channel handle. This channel handle is required as parameter to other ACE driver functions dealing with analog inputs. It will return INVALID_CHANNEL_HANDLE if the channel ID passed as parameter is invalid.

ACE_get_channel_count

Prototype

```
uint32_t ACE_get_channel_count
(
  void
);
```

Description

The *ACE_get_channel_count()* function returns the total number of configured analog input channels. It is the number of channels available for use as opposed to the theorical number of physical channels supported by the device.

Return Value

The *ACE_get_channel_count()* function returns the total number of input channels that were configured in the ACE configurator. The *ACE_get_channel_count()* function returns 0 if no input channels were configured.

Example

```
uint32_t inc;
uint32_t nb_of_channels;
ace_channel_handle_t current_channel;

nb_of_channels = ACE_get_channel_count();
current_channel = ACE_get_first_channel();

for (inc = 0; inc < nb_of_channels; ++inc)
{
    adc_result = ACE_get_ppe_sample( current_channel );
    display_value( current_channel, adc_result );
    current_channel = ACE_get_next_channel( current_channel );
}
```

ACE_get_first_channel

Prototype

```
ace_channel_handle_t ACE_get_first_channel
(
  void
);
```

Description

The *ACE_get_first_channel()* function returns the channel handle of one of the channel controlled by the ACE. This function is used to start iterating though the list of analog input channels handled by the ACE.

Parameters

This function takes no parameters.

Return Value

The *ACE_get_first_channel()* function returns the first channel handle found in the ACE driver's internal channel handles list or INVALID_CHANNEL_HANDLE if there are no channels defined in the ACE configuration.

Example

```
uint32_t inc;
uint32_t nb_of_channels;
ace_channel_handle_t current_channel;

nb_of_channels = ACE_get_channel_count();
current_channel = ACE_get_first_channel();

for (inc = 0; inc < nb_of_channels; ++inc)
{
    adc_result = ACE_get_ppe_sample( current_channel );
    display_value( current_channel, adc_result );
    current_channel = ACE_get_next_channel( current_channel );
}
```

ACE_get_next_channel

Prototype

```
ace_channel_handle_t ACE_get_next_channel
(
  ace_channel_handle_t channel_handle
)
```

Description

The *ACE_get_next_channel()* returns the channel handle of the channel following the one passed as parameter. This function is used to iterate through the list of analog input channels handled by the ACE.

Parameters

channel_handle

The *channel_handle* parameter identifies from which channel the driver should look in its channel handle list to return the next channel handle. The *channel_handle* parameter would typically be the channel handle returned by a call to *ACE_get_first_channel()* or a previous call to *ACE_get_next_channel()*.

Note: The first call to *ACE_get_next_channel()* would typically use the *channel_handle* returned by a previous call to *ACE_get_first_channel()*. The second and subsequent calls to *ACE_get_next_channel()* would typically use the *channel_handle* returned by a previous call to *ACE_get_next_channel()*.

Return Value

The *ACE_get_next_channel()* function returns the channel handle of the channel following the one passed as parameter or INVALID_CHANNEL_HANDLE if the end of the channels list has been reached.

Example

```
uint32_t inc;
uint32_t nb_of_channels;
ace_channel_handle_t current_channel;

nb_of_channels = ACE_get_channel_count();
current_channel = ACE_get_first_channel();

for (inc = 0; inc < nb_of_channels; ++inc)
{
    adc_result = ACE_get_ppe_sample( current_channel );
    display_value( current_channel, adc_result );
    current_channel = ACE_get_next_channel( current_channel );
}
```

ACE_get_ppe_sample

Prototype

```
uint16_t ACE_get_ppe_sample
(
  ace_channel_handle_t channel_handle
);
```

Description

The *ACE_get_ppe_sample()* function is used to read the most recent post processed sample for the analog input channel associated with the channel handle passed as parameter.

Parameters

channel_handle

The *channel_handle* parameter identifies the analog input channel for which this function will return the most recent ADC conversion result adjusted for calibration and user provided coefficients as provided through the ACE configurator. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

Return Value

This function returns a 16 bit value representing the adjusted value of the ADC conversion result for the analog input channel identified by the channel handle passed as parameter. The return value is actually a 12, 10 or 8 bits number depending on the configuration of the ADC.

Example

```
uint16_t adc_result;
ace_channel_handle_t at0;
at0 = ACE_get_channel_handle("VoltageMonitorAT0");
adc_result = ACE_get_ppe_sample( at0 );
```

ACE_get_channel_name

Prototype

```
const uint8_t * ACE_get_channel_name
(
  ace_channel_handle_t channel_handle
);
```

Description

The *ACE_get_channel_name()* function provides the name of the channel associated with the channel handle passed as parameter. The channel name is the name used in the ACE configurator software tool when adding a service to the ACE.

Parameters

channel_handle

The *channel_handle* parameter identifies the analog input channel for which we want to retrieve the channel name. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

Return Value

This function returns a pointer to a zero-terminated string containing the name of the channel. It returns 0 if the channel handle passed as parameter is not recognized.

ACE_get_channel_type

Prototype

```
channel_type_t ACE_get_channel_type
(
  ace_channel_handle_t channel_handle
);
```

Description

The *ACE_get_channel_type()* function returns the type of input channel of the analog input channel identified by the channel handle passed as parameter. This function allows determining whether the quantity measured through the ADC is a voltage, current or temperature.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

Return Value

This function returns one of the following values to report the type of quantity measured through the channel:

- VOLTAGE
- CURRENT
- TEMPERATURE

Functions − Post Processing Engine Flags

The following functions are used to control interrupts generated by the ACE's PPE when monitored inputs rise above or fall below thresholds specified in the ACE configurator software tool.

- *ACE_get_flag_handle()*
- *ACE_get_channel_flag_count()*
- *ACE_get_channel_first_flag()*
- *ACE_get_channel_next_flag()*
- *ACE_get_flag_status()*
- *ACE_register_flag_isr()*
- *ACE_enable_flag_irq()*
- *ACE_disable_flag_irq()*
- *ACE_clear_flag_irq()*
- *ACE_register_channel_flags_isr()*
- *ACE_enable_channel_flags_irq()*
- *ACE_disable_channel_flags_irq()*
- *ACE_clear_channel_flags_irq()*
- *ACE_register_global_flags_isr()*
- *ACE_get_flag_name()*
- *ACE_get_flag_channel()*
- *ACE_is_hysteresis_flag()*
- *ACE_is_under_flag()*

The following functions are used to dynamically control PPE flags threshholds.

- *ACE_set_flag_threshold()*
- *ACE_set_flag_hysteresis()*
- *ACE_set_flag_assertion()*
- *ACE_set_flag_deassertion()*
- *ACE_set_channel_hysteresis()*

ACE_get_flag_handle

Prototype

```
ace_flag_handle_t ACE_get_flag_handle
(
  const uint8_t * p_sz_full_flag_name
);
```

Description

The *ACE_get_flag_handle()* function returns the handle of the flag identified by the flag name passed as parameter. The flag handle obtained through this function is then used as parameter to other flag control functions to identify which flag is to be controlled by the called function.

Parameters

p_sz_full_flag_name

The *p_sz_full_flag_name* parameter is a pointer to a zero-terminated string holding the name of the flag as specified in the ACE configurator. The full name of a flag contains both the name of the monitored input channel and the name of the flag generated based the level of that input separated by ":". For example, the full name for the flag called "CriticalOver" raised when the input channel called "MainSupply" reaches a critical level would be named "MainSupply:CriticalOver".

Return Value

The *ACE_get_flag_handle()* returns the flag handle associated with the flag name passed as parameter. It returns INVALID_FLAG_HANDLE when the flag name is invalid and not recognized by the ACE driver.

ACE_get_channel_flag_count

Prototype

```
uint32_t ACE_get_channel_flag_count
(
        ace_channel_handle_t channel_handle
);
```

Description

The *ACE_get_channel_flag_count()* function returns the total number of flags associated with the input channel specified by the *channel_handle* parameter. It indicates how many flags are generated based on the value of the specified analog input channel.

Parameters:

channel_handle

The channel_handle parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

Returns:

The *ACE_get_channel_flag_count***()** function returns the total number of flags that are generated based on the value of the specified analog input channel. The *ACE_get_channel_flag_count***()** function returns 0 if no input channels were configured.

Example

```
uint32_t inc;
uint32_t nb_of_flags;
uint16_t flag_iterator;
ace_flag_handle_t current_flag;
ace_channel_handle_t channel_handle;

nb_of_flags = ACE_get_channel_flag_count(channel_handle);
current_flag = ACE_get_channel_first_flag(channel_handle, &flag_iterator);

for (inc = 0; inc < nb_of_flags; ++inc)
{
    current_flag = ACE_get_channel_next_flag(channel_handle, &flag_iterator);
    display_flag_properties(current_flag);
}
```

ACE_get_channel_first_flag

Prototype

```
ace_flag_handle_t ACE_get_channel_first_flag
(
  ace_channel_handle_t channel_handle,
  uint16_t * iterator
);
```

Description

The *ACE_get_channel_first_flag()* function retrieves the handle of the first flag associated with the analog input channel identified by the channel handle passed as parameter. It also initializes the value of the iterator variable pointed to by the second function parameter. The iterator can be used subsequently as a parameter to the *ACE_get_channel_next_flag()* function to iterate through all flags associated with the analog input channel.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

Iterator

The *iterator* parameter is a pointer to a uint16_t iterator variable. The value of the iterator variable will be set by the *ACE_get_channel_first_flag()* functions so that it can be used in subsequent calls to *ACE_get_channel_next_flag()* to keep track of the current location in the list of flags associated with the analog input channel.

Return Value

The *ACE_get_channel_first_flag()* function returns a flag handle identifying one of the flags generated based on the value of the analog input channel identified by the *channel_handle* parameter. It returns INVALID_FLAG_HANDLE if no flags are generated based on the analog input channel input or if the channel handle is invalid.

Example

```
uint32_t inc;
uint32_t nb_of_flags;
uint16_t flag_iterator;
ace_flag_handle_t current_flag;
ace_channel_handle_t channel_handle;

nb_of_flags = ACE_get_channel_flag_count(channel_handle);
current_flag = ACE_get_channel_first_flag(channel_handle, &flag_iterator);

for (inc = 0; inc < nb_of_flags; ++inc)
{
    current_flag = ACE_get_channel_next_flag(channel_handle, &flag_iterator);
    display_flag_properties(current_flag);
}
```

ACE_get_channel_next_flag

Prototype

```
ace_flag_handle_t ACE_get_channel_next_flag
(
  ace_channel_handle_t channel_handle,
  uint16_t * iterator
);
```

Description

The *ACE_get_channel_next_flag()* function retrieves the handle of a flag associated with the analog input channel identified by the channel handle passed as parameter. The retrieved flag handle is the next one in the driver's internal flag list based on the iterator parameter.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

iterator

The *iterator* parameter is a pointer to a uint16_t iterator variable. The value of the iterator variable will be set by the *ACE_get_channel_first_flag()* functions so that it can be used in subsequent calls to *ACE_get_channel_next_flag()* to keep track of the current location in the list of flags associated with the analog input channel.

Return Value

The *ACE_get_channel_next_flag()* function returns a flag handle identifying one of the flags generated based on the value of the analog input channel identified by the *channel_handle* parameter. It returns INVALID_FLAG_HANDLE if no flags are generated based on the analog input channel input or if the channel handle is invalid.

Example

```
uint32_t inc;
uint32_t nb_of_flags;
uint16_t flag_iterator;
ace_flag_handle_t current_flag;
ace_channel_handle_t channel_handle;

nb_of_flags = ACE_get_channel_flag_count(channel_handle);
current_flag = ACE_get_channel_first_flag(channel_handle, &flag_iterator);

for (inc = 0; inc < nb_of_flags; ++inc)
{
    current_flag = ACE_get_channel_next_flag(channel_handle, &flag_iterator);
    display_flag_properties(current_flag);
}
```

ACE_get_flag_status

Prototype

```
int32_t ACE_get_flag_status
(
  ace_flag_handle_t flag_handle
);
```

Description

The *ACE_get_flag_status()* function returns the current status of the flag specified as parameter. The flag is identified through the name specified in the ACE configurator when the flag was created.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

Return Value

The *ACE_get_flag_status()* function returns one of the following values depending on the current status of the flag:

- FLAG_ASSERTED – if the flag is raised/asserted.
- FLAG_NOT_ASSERTED – if the flag is not asserted.
- UNKNOWN_FLAG – if the flag name is not recognized by the driver.

ACE_register_flag_isr

Prototype

```
void ACE_register_flag_isr
(
  ace_flag_handle_t flag_handle,
  flag_isr_t flag_isr
);
```

Description

The *ACE_register_flag_isr()* function is used to register a handler function with the ACE driver. The registered function will be called by the ACE driver when the associated flag is raised by the ACE post processing engine.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

flag_isr

The *flag_isr* parameter is a pointer to a flag handler function with the following prototype:

void handler_function_name(ace_flag_handle_t flag_handle)

The flag handler function is called by the ACE driver as part of the relevant post processing engine flag interrupt service routine. It does not need to handle flag interrupt clearing as this is done by the ACE driver.

Return Value

This function does not return a value.

Example

```
void my_critical_handler( ace_flag_handle_t flag_handle );

void system_init( void )
{
    ace_flag_handle_t flag_handle;

    flag_handle = ACE_get_flag_handle( "MainSupply:CriticalLevel" );
    ACE_register_flag_isr( flag_handle, my_critical_handler );
    ACE_enable_flag_irq( flag_handle );
}

void my_critical_handler(ace_flag_handle_t flag_handle )
{
    panic( flag_handle );
}
```

ACE_enable_flag_irq

Prototype

```
void ACE_enable_flag_irq
(
  ace_flag_handle_t flag_handle
);
```

Description

The *ACE_enable_flag_irq()* function enables the ACE PPE generated flags specified as parameter to interrupt the Cortex-M3 processor. It enables flag interrupts both at the ACE PPE flag and Cortex-M3 interrupt controller levels.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

Return Value

This function does not return a value.

ACE_disable_flag_irq

Prototype

```
void ACE_disable_flag_irq
(
  ace_flag_handle_t flag_handle
);
```

Description

The *ACE_disable_flag_irq()* function disables ACE PPE generated flags from interrupting the Cortex-M3. The interrupt is only disabled at the ACE PPE flag level in order to avoid disabling other flags interrupts which may happen to use the same ACE threshold interrupt line.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

Return Value

This function does not return a value.

ACE_clear_flag_irq

Prototype

```
void ACE_clear_flag_irq
(
  ace_flag_handle_t flag_handle
);
```

Description

The *ACE_clear_flag_irq()* function clears the interrupt for the flag specified as parameter. This function would typically be used before enabling the flag interrupt in order to ignore past events.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

Return Value

This function does not return a value.

ACE_register_channel_flags_isr

Prototype

```
void ACE_register_channel_flags_isr
(
  ace_channel_handle_t channel_handle,
  channel_flag_isr_t channel_flag_isr
);
```

Description

The *ACE_register_channel_flags_isr()* function is used to register a flag interrupt handler function with the ACE driver. The registered interrupt handler will be called by the ACE driver when one of the flag generated based on the value of the analog input channel identified by the channel handle passed as parameter is asserted.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

channel_flag_isr

The *channel_flag_isr* parameter is pointer to a function taking a flag handle as parameter.

void handler_function_name(ace_flag_handle_t flag_handle)

The flag handler function is called by the ACE driver as part of the relevant post processing engine flag interrupt service routine. It does not need to handle flag interrupt clearing as this is done by the ACE driver.

Return Value

This function does not return a value.

Example

The example below demonstrates the use of the ACE_register_channel_flags_isr() function in a system where the ACE is configured to have an analog input channels named "MainSupply" with one flag named "Critical" generated based on the value of "MainSupply" channel. The names "MainSupply" and "Critical" were user selected in the ACE configurator.

```
void main_supply_handler (ace_flag_handle_t flag_handle);

void system_init(void)
{
    ACE_register_channel_flag_isr(MainSupply, main_supply_handler);
    ACE_enable_channel_flags_irq(MainSupply);
}

void main_supply_handler (ace_flag_handle_t flag_handle)
{
    if (MainSupply_Critical == flag_handle)
    {
        panic(flag_handle);
    }
}
```

ACE_enable_channel_flags_irq

Prototype

```
void ACE_enable_channel_flags_irq
(
  ace_channel_handle_t channel_handle
);
```

Description

The *ACE_enable_channel_flags_irq()* function enables all flags generated based on the value of the analog input channel passed as parameter to generate interrupts. Flags used to detect that thresholds are crossed by the value sampled on the analog input channel identified as parameter are enabled to generate interrupts by this function. It enables flag interrupts both at the ACE PEE flag and Cortex-M3 interrupt controller levels.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

Return Value

This function does not return a value.

ACE_disable_channel_flags_irq

Prototype

```
void ACE_disable_channel_flags_irq
(
  ace_channel_handle_t channel_handle
);
```

Description

The *ACE_disable_channel_flags_irq()* function disables all flags generated based on the value of the analog input channel passed as parameter to generate interrupts. Flags used to detect that thresholds are crossed by the value sampled on the analog input channel identified as parameter are disabled from generating interrupts by this function. The interrupt is only disabled at the ACE PPE flag level in order to avoid disabling other channel's flag interrupts which may happen to use the same ACE threshold interrupt line.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

Return Value

This function does not return a value.

ACE_clear_channel_flags_irq

Prototype

```
void ACE_clear_channel_flags_irq
(
  ace_channel_handle_t channel_handle
);
```

Description

The *ACE_clear_channel_flags_irq()* function clears all interrupts generated by flags associated with the analog input channel passed as parameter. Interrupt generated by flags used to detect that thresholds are crossed by the value sampled on the analog input channel identified as parameter are cleared by this function. This function would typically be used before enabling the flag interrupts in order to ignore past events.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

Return Value

This function does not return a value.

ACE_register_global_flags_isr

Prototype

```
void ACE_register_global_flags_isr
(
  global_flag_isr_t global_flag_isr
);
```

Description

The *ACE_register_global_flags_isr()* function is used to register a global flag handler function with the ACE driver. The registered global handler will be called when any flag interrupt is generated.

Parameters

global_flag_isr

The *global_flag_isr* parameter is a pointer to a function taking a flag handle and channel handle as parameter.

Return Value

This function does not return a value.

ACE_get_flag_name

Prototype

```
const uint8_t * ACE_get_flag_name
(
  ace_flag_handle_t flag_handle
);
```

Description

The *ACE_get_flag_name()* function returns the name of the flag identified by the flag handle passed as parameter.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

Return Value

The *ACE_get_flag_name()* function returns a pointer to a zero-terminated string containing the name of the flag identified by the flag handle passed as parameter. It returns 0 if the flag handle passed as parameter is invalid.

ACE_get_flag_channel

Prototype

```
ace_channel_handle_t ACE_get_flag_channel
(
  ace_flag_handle_t flag_handle
);
```

Description

The *ACE_get_flag_channel()* function returns the handle of the channel monitored in order to generate the flag identified by the flag handle passed as parameter.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

Return Value

The *ACE_get_flag_channel()* function returns a channel handle identifying the analog input channel monitored by the flag passed as parameter.

ACE_is_hysteresis_flag

Prototype

```
uint32_t ACE_is_hysteresis_flag
(
    ace_flag_handle_t   flag_handle
);
```

Description

The *ACE_is_hysteresis_flag()* function indicates if an hysteresis is applied to the analog input sample value when determining the state of the flag identified as parameter.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

Return Value

This function returns the value one if a hysteresis is applied to the channel sample values as part of determining the state of the flag identified as parameter. It returns zero if no hysteresis is applied.

ACE_is_under_flag

Prototype

```
uint32_t ACE_is_under_flag
(
        ace_flag_handle_t flag_handle
);
```

Description

The *ACE_is_under_flag()* function indicates whether a flag is triggered when the monitored analog input falls below the flag's threshold level or above the flag's threshold level.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

Return Value

This function returns the value one if the flag identified as parameter triggers as a result of the monitored input falling below the flag's threshold value. It returns zero if the flag triggers as a result of the monitored input exceeding the flag's threshold value.

ACE_set_flag_threshold

Prototype

```
void ACE_set_flag_threshold
(
  ace_flag_handle_t flag_handle,
  uint16_t new_threshold
);
```

Description

The *ACE_set_flag_threshold()* function is used to adjust the threshold for a specific post processing engine generated flag. The flag is identified through the name selected in the ACE configurator software tool.

This function will set a new flag's threshold value while preserving the hysteresis specified at configuration time or through a call to *ACE_set_flag_hysteresis()*. For example, requesting a 1 volt threshold for an over flag configured with a 100 millivolts hysteresis will result in the flag being asserted when the voltage reaches 1.1 volts and deasserted when the voltage falls below 0.9 volt.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

new_threshold

The *new_threshold* parameter specifies the new threshold level that must be reached in order for the flag to be raised. The value of this parameter is the sample value resulting from a post processing engine conversion of the desired analog input threshold level.

Return Value

This function does not return a value.

Example

The function below sets the threshold of the flag specified as parameter to 1 volt.

```
void set_threshold_to_1V
(
    ace_flag_handle_t flag_handle
)
{
    uint16_t new_threshold;
    ace_channel_handle_t channel_handle;

    channel_handle = ACE_get_flag_channel(flag_handle);
    new_threshold = ACE_convert_from_mV(channel_handle, 1000);
    ACE_set_flag_threshold(flag_handle, new_threshold);
}
```

ACE_set_flag_hysteresis

Prototype

```
void ACE_set_flag_hysteresis
(
  ace_flag_handle_t flag_handle,
  uint16_t adc_hysteresis
);
```

Description

The *ACE_set_flag_hysteresis()* function modifies the hysteresis applied to the analog input channel sample values used to generate the flag specified as parameter.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

adc_hysteresis

The *adc_hysteresis* parameter is the value to add and subtract to the threshold value to obtain the hysteresis high and low limits triggering flag assertion and deassertion. The *adc_hysteresis* parameter is a PPE conversion result offset.

Return Value

This function does not return a value.

Example

The example below demonstrates the use of the ACE_set_flag_hysteresis() function to set a 100mV hysteresis on the OVER_1V flag of the VoltageMonitor input channel. VoltageMonitor and OVER_1V are names selected in the ACE configurator for one of the analog inputs and one of the flags associated with that input.

The method used to compute the adc_hysteresis value will work for all input types including ABPS inputs where zero Volts is not equivalent to a PPE sample value of zero.

```
ace_channel_handle_t channel_handle;
ace_flag_handle_t flag_handle;
uint16_t adc_hysteresis;
uint16_t upper_limit;
uint16_t lower_limit;

channel_handle = VoltageMonitor;
flag_handle = VoltageMonitor_OVER_1V;

upper_limit = ACE_convert_from_mV(channel_handle, 100);
lower_limit = ACE_convert_from_mV(channel_handle, 0);

if (upper_limit > lower_limit)
{
    adc_hysteresis = upper_limit - lower_limit;
}
else
{
    adc_hysteresis = lower_limit - upper_limit;
}

ACE_set_flag_hysteresis(flag_handle, adc_hysteresis);
```

ACE_set_flag_assertion

Prototype

```
void ACE_set_flag_assertion
(
  ace_flag_handle_t flag_handle,
  uint16_t assertion_value
)
```

Description

The *ACE_set_flag_assertion()* function sets the PPE sample value that must be reached in order for the flag specified as parameter to become asserted. It is used in conjunction with the *ACE_set_flag_deassertion()* function as an alternative to the *ACE_set_flag_threshold()* and *ACE_set_flag_hysteresis()* functions to set the hysteresis window of an hysteresis flag.

The *ACE_set_flag_assertion()* and *ACE_set_flag_deassertion()* functions are intended to be used where the threshold value is not centered within the hysteresis window. They allow specifying the actual threshold values at which the flag will be asserted and deasserted.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

assertion_value

The *assertion_value* parameter is the PPE sample value that must be reached for the flag, identified through the *flag_handle* parameter, to become asserted. The PPE sample value is always a 12 bits sample value regardless of the configuration of the ADC used to sample the input channel.

Return Value

This function does not return a value.

ACE_set_flag_deassertion

Prototype

```
void ACE_set_flag_deassertion
(
  ace_flag_handle_t flag_handle,
  uint16_t assertion_value
)
```

Description

The *ACE_set_flag_deassertion()* function sets the PPE sample value that must be reached in order for the flag specified as parameter to become deasserted. It is used in conjunction with the *ACE_set_flag_assertion()* function as an alternative to the *ACE_set_flag_threshold()* and *ACE_set_flag_hysteresis()* functions to set the hysteresis window of an hysteresis flag.

The *ACE_set_flag_assertion()* and *ACE_set_flag_deassertion()* functions are intended to be used where the threshold value is not centered within the hysteresis window. They allow specifying the actual threshold values at which the flag will be asserted and deasserted.

Parameters

flag_handle

The *flag_handle* parameter identifies one of the flags generated based on the value of an analog input channel. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The flag handle value can also be retrieved through a call to *ACE_get_flag_handle()* when the name of the flag is known, or by iterating though all flags associated with an analog input channel using the *ACE_get_channel_first_flag()* and *ACE_get_channel_next_flag()*.

assertion_value

The *assertion_value* parameter is the PPE sample value that must be reached for the flag, identified through the *flag_handle* parameter, to become deasserted. The PPE sample value is always a 12 bits sample value regardless of the configuration of the ADC used to sample the input channel.

Return Value

This function does not return a value.

ACE_set_channel_hysteresis

Prototype

```
void ACE_set_channel_hysteresis
(
  ace_channel_handle_t channel_handle,
  uint16_t adc_hysteresis
);
```

Description

The *ACE_set_channel_hysteresis()* function sets the hysteresis applied to analog input channel sample values when generating flags. It sets the hysteresis for all flags generated based on the value of the analog input channel identified by the channel handle passed as first parameter.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

adc_hysteresis

The *adc_hysteresis* parameter is the value to add and subtract to the threshold value to obtain the hysteresis high and low limits triggering flag assertion and deassertion. The *adc_hysteresis* parameter is a PPE conversion result offset.

Return Value

This function does not return a value.

Functions − Conversion

The following functions are used to convert an ADC sample value to a real world unit.

- *ACE_convert_adc_input_to_mV()*
- *ACE_convert_to_mV()*
- *ACE_convert_to_mA()*
- *ACE_convert_to_Kelvin()*
- *ACE_convert_to_Celsius()*
- *ACE_convert_to_Fahrenheit()*

The following functions are used to convert a real world unit to an ADC sample value.

- *ACE_convert_mV_to_adc_value()*
- *ACE_convert_from_mV()*
- *ACE_convert_from_mA()*
- *ACE_convert_from_Kelvin()*
- *ACE_convert_from_Celsius()*
- *ACE_convert_from_Fahrenheit()*

The following function is used to translate a PDMA sample value into ADC sample value and channel.

- *ACE_translate_pdma_value()*

ACE_convert_adc_input_to_mV

Prototype

```
uint32_t ACE_convert_adc_input_to_mV
(
  ace_channel_handle_t channel_handle,
  uint16_t sample_value
);
```

Description

The *ACE_convert_adc_input_to_mV()* function converts an ADC sample value into the value in millivolts of the voltage seen at ADC input. It does not account for prescaling taking place before the ADC hardware input.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available flag handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

sample_value

The *sample_value* parameter is the result of an analog to digital conversion.

Return Value

The *ACE_convert_adc_input_to_mV()* returns the number of millivolts derived from the ADC sample value passed as parameter.

ACE_convert_to_mV

Prototype

```
int32_t ACE_convert_to_mV
(
  ace_channel_handle_t channel_handle,
  uint16_t sample_value
);
```

Description

The *ACE_convert_to_mV()* function converts a PPE sample value into millivolts. It handles prescaling adjustments based on ACE configuration for ABPS inputs.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

sample_value

The *sample_value* parameter is the result of an analog to digital conversion.

Return Value

The *ACE_convert_to_mV()* returns the number of millivolts derived from the PPE sample value passed as parameter. The returned value can be either positive or negative since prescaling is accounted for in the conversion.

ACE_convert_to_mA

Prototype

```
uint32_t ACE_convert_to_mA
(
  ace_channel_handle_t channel_handle,
  uint16_t sample_value
);
```

Description

The *ACE_convert_to_mA()* function converts a PPE sample value into milliamps. The result of the conversion is only meaningful if the PPE sample value results from the conversion of a current monitor input.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

sample_value

The *sample_value* parameter is the result of an analog to digital conversion of the voltage generated by a current monitor analog block.

Return Value

The *ACE_convert_to_mA()* returns the number of milliamps derived from the PPE sample value passed as parameter.

ACE_convert_to_Kelvin

Prototype

```
uint32_t ACE_convert_to_Kelvin
(
  ace_channel_handle_t channel_handle,
  uint16_t sample_value
);
```

Description

The *ACE_convert_to_Kelvin()* function converts a PPE sample value into degrees Kelvin. The result of the conversion is only meaningful if the PPE sample value results from the conversion of a temperature monitor input.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

sample_value

The *sample_value* parameter is the result of an analog to digital conversion of the voltage generated by a temperature monitor analog block.

Return Value

The *ACE_convert_to_Kelvin()* returns the number of degrees Kelvin derived from the PPE sample value passed as parameter.

ACE_convert_to_Celsius

Prototype

```
uint32_t ACE_convert_to_Celsius
(
  ace_channel_handle_t channel_handle,
  uint16_t sample_value
);
```

Description

The *ACE_convert_to_Celsius()* function converts a PPE sample value into tenths of degrees Celsius. The result of the conversion is only meaningful if the PPE sample value results from the conversion of a temperature monitor input.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

sample_value

The *sample_value* parameter is the result of an analog to digital conversion of the voltage generated by a temperature monitor analog block.

Return Value

The *ACE_convert_to_Celsius()* returns the number of tenths of degrees Celsius derived from the PPE sample value passed as parameter.

ACE_convert_to_Fahrenheit

Prototype

```
uint32_t ACE_convert_to_Fahrenheit
(
  ace_channel_handle_t channel_handle,
  uint16_t sample_value
);
```

Description

The *ACE_convert_to_Fahrenheit()* function converts a PPE sample value into degrees Fahrenheit. The result of the conversion is only meaningful if the PPE sample value results from the conversion of a temperature monitor input.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

sample_value

The *sample_value* parameter is the result of an analog to digital conversion of the voltage generated by a temperature monitor analog block.

Return Value

The *ACE_convert_to_Fahrenheit()* returns the number of degrees Fahrenheit derived from the PPE sample value passed as parameter.

ACE_convert_mV_to_adc_value

Prototype

```
uint32_t ACE_convert_mV_to_adc_value
(
  ace_channel_handle_t channel_handle,
  uint32_t voltage
);
```

Description

The *ACE_convert_mV_to_adc_value()* function converts a voltage value given in millivolts into the ADC sample value that would result from sampling this voltage value on the analog input channel specified as parameter. This function is intended for use when directly controlling the ADC, not when using samples read from the PPE. It does not account for prescaling taking place before the ADC hardware input.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

voltage

The *voltage* parameter is the mill volts voltage value for which we want this function to return the associated ADC sample result value.

Return Value

The ACE_convert_mV_to_adc_value() returns the ADC sample value that would be produced if the analog input channel identified by channel_handle was set to the voltage value passed as second parameter.

ACE_convert_from_mV

Prototype

```
uint16_t ACE_convert_from_mV
(
  ace_channel_handle_t channel_handle,
  int32_t voltage
);
```

Description

The *ACE_convert_from_mV()* function converts a voltage value given in millivolts into the PPE sample value that would result from sampling this voltage value on the analog input channel specified as parameter. This function handles prescaling adjusments based on ACE configuration for ABPS inputs.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

voltage

The *voltage* parameter is the millivolts voltage value for which we want this function to return the associated PPE sample result value.

Return Value

The *ACE_convert_from_mV()* returns the PPE sample value that would be produced if the analog input channel identified by channel_handle was set to the voltage value passed as second parameter.

ACE_convert_from_mA

Prototype

```
uint16_t ACE_convert_from_mA
(
  ace_channel_handle_t channel_handle,
  uint32_t current
)
```

Description

The *ACE_convert_from_mA()* function converts a current value given in milliamps into the PPE sample value that would result from sampling this current value on the analog input channel specified as parameter. The result of the conversion is only meaningful if the analog input channel specified as parameter is configured as a current monitoring channel.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

current

The *current* parameter is the milliamps current value for which we want this function to return the associated PPE sample result value.

Return Value

The *ACE_convert_from_mA()* returns the PPE sample value that would be produced if the analog input channel identified by channel_handle was set to the current value passed as second parameter.

ACE_convert_from_Kelvin

Prototype

```
uint16_t ACE_convert_from_Kelvin
(
  ace_channel_handle_t channel_handle,
  uint32_t temperature
)
```

Description

The *ACE_convert_from_Kelvin()* function converts a temperature value given in degrees Kelvin into the PPE sample value that would result from sampling this temperature value on the analog input channel specified as parameter. The result of the conversion is only meaningful if the analog input channel specified as parameter is configured as a temperature monitoring channel.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

temperature

The *temperature* parameter is the degrees Kelvin temperature value for which we want this function to return the associated PPE sample result value.

Return Value

The *ACE_convert_from_Kelvin()* returns the PPE sample value that would be produced if the analog input channel identified by channel_handle was set to the temperature value passed as second parameter.

ACE_convert_from_Celsius

Prototype

```
uint16_t ACE_convert_from_Celsius
(
  ace_channel_handle_t channel_handle,
  int32_t temperature
)
```

Description

The *ACE_convert_from_Celsius()* function converts a temperature value given in degrees Celsius into the PPE sample value that would result from sampling this temperature value on the analog input channel specified as parameter. The result of the conversion is only meaningful if the analog input channel specified as parameter is configured as a temperature monitoring channel.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

temperature

The *temperature* parameter is the degrees Celsius temperature value for which we want this function to return the associated PPE sample result value.

Return Value

The *ACE_convert_from_Celsius()* returns the PPE sample value that would be produced if the analog input channel identified by channel_handle was set to the temperature value passed as second parameter.

ACE_convert_from_Fahrenheit

Prototype

```
uint16_t ACE_convert_from_Fahrenheit
(
  ace_channel_handle_t channel_handle,
  int32_t temperature
)
```

Description

The *ACE_convert_from_Fahrenheit()* function converts a temperature value given in degrees Fahrenheit into the PPE sample value that would result from sampling this temperature value on the analog input channel specified as parameter. The result of the conversion is only meaningful if the analog input channel specified as parameter is configured as a temperature monitoring channel.

Parameters

channel_handle

The *channel_handle* parameter identifies one of the analog input channels monitored by the ACE. The available channel handle values can be found in the ace_handles.h file located in the .\drivers_config\mss_ace subdirectory. The channel handle value can also be retrieved through a call to *ACE_get_channel_handle()* when the name of the channel is known, or by iterating though all analog input channel using the *ACE_get_first_channel()* and *ACE_get_next_channel()*.

temperature

The *temperature* parameter is the degrees Fahrenheit temperature value for which we want this function to return the associated PPE sample result value.

Return Value

The *ACE_convert_from_Fahrenheit()* returns the PPE sample value that would be produced if the analog input channel identified by channel_handle was set to the temperature value passed as second parameter.

ACE_translate_pdma_value

Prototype

```
uint16_t ACE_translate_pdma_value
(
        uint32_t pdma_value,
        adc_channel_id_t * channel_id
);
```

Description

The *ACE_translate_pdma_value()* function translates PDMA sampling values, received from the ACE via PDMA transfers, into input channel ID and PPE sample value. The PDMA sampling values are generated by the ACE as a result of selecting "Send result to DMA" in the ACE configurator analog input configuration. The PDMA sampling values can be either raw ADC values, filtered values or the result of a linear transformation. The PDMA sampling values are obtained by configuring the PDMA controller to transfer data from the ACE into a memory buffer. The *ACE_translate_pdma_value()* function is used to interpret the content of that memory buffer.

Note: The translation of PDMA data containing raw ADC values from ABPS inputs will result in sample values with an unexpected polarity. The returned sample value will have the opposite polarity to the actual analog value seen on the ABPS input. This is due to the internal characteristics of the analog front end that are normally hidden by the PPE processing of ADC raw samples. The translation of raw ADC values from analog inputs other than ABPS inputs will result in correct sample values.

Parameters

pdma_value

The *pdma_value* parameter is a 32-bit value received from the ACE through a peripheral DMA transfer.

channel_id

The *channel_id* parameter is a pointer to an ADC channel ID variable. It is used to return the ID of the ADC channel from which the PPE sample value was generated from. This parameter can be set to zero if retrieving the channel ID is not required.

Return Value

The *ACE_translate_pdma_value()* returns the PPE sample value extracted from the PDMA sampling value.

Example

```
uint16_t ppe_value;
uint16_t pdma_value;
adc_channel_id_t channel_id;
ace_channel_handle_t channel_handle;

pdma_value = get_next_pdma_ace_sample();
ppe_value = ACE_translate_pdma_value(pdma_value, &channel_id);
channel_handle = ACE_get_input_channel_handle(channel_id);
if (channel_handle != INVALID_CHANNEL_HANDLE)
{
    display_value(channel_handle, ppe_value);
}
```

Functions − Sample Sequencing Engine Control

The following functions are used to control the ACE Sample Sequencing Engine (SSE).

- *ACE_get_sse_seq_handle()*
- *ACE_load_sse()*
- *ACE_start_sse()*
- *ACE_restart_sse()*
- *ACE_stop_sse()*
- *ACE_resume_sse()*

ACE_get_sse_seq_handle

Prototype

```
sse_sequence_handle_t ACE_get_sse_seq_handle
(
  const uint8_t * p_sz_sequence_name
)
```

Description

The *ACE_get_sse_seq_handle()* function retrieves the handle of the Sample Sequencing Engine sequence identified by the sequence name passed as parameter. The sequence handle can then be used as parameter to other SSE sequence control functions to identify the sequence to control.

Parameters

p_sz_sequence_name

The *p_sz_sequence*_name parameter is a pointer to a zero-terminated string containing the name of the sampling sequence for which we want to retrieve the handle.

Return Value

The *ACE_get_sse_seq_handle()* function returns the handle used to identify the sequence passed as parameter with other ACE driver sampling sequence control functions. It returns INVALID_SSE_SEQ_HANDLE if the sequence name passed as parameter is not recognized.

Example

```
sse_sequence_handle_t sse_seq_handle;
sse_seq_handle = ACE_get_sse_seq_handle("ProcedureA");
if ( sse_seq_handle != INVALID_SSE_SEQ_HANDLE )
{
    ACE_load_sse( sse_seq_handle );
    ACE_start_sse( sse_seq_handle );
}
```

ACE_load_sse

Prototype

```
void ACE_load_sse
(
  sse_sequence_handle_t sequence
)
```

Description

The *ACE_load_sse()* function loads the ACE SSE RAM with the microcode implementing the sampling sequence identified by the SSE sequence handle passed as parameter.

Parameters

sequence

The *sequence* parameter is the SSE sequence handle identifying the sampling sequence to load into the ACE SSE. The value for this handle is retrieved through a call to function *ACE_get_sse_seq_handle()*.

Return Value

This function does not return a value.

ACE_start_sse

Prototype

```
void ACE_start_sse
(
  sse_sequence_handle_t sequence
)
```

Description

The *ACE_start_sse()* function causes the SSE to start executing the sequence identified by the sequence handle passed as parameter. It causes the initialization part of the sampling sequence to be executed before the loop part of the sequence is started. You must ensure that the sampling sequence has been loaded into the ACE's SSE before calling this function.

Parameters

sequence

The *sequence* parameter is the SSE sequence handle identifying the sampling sequence to load into the ACE SSE. The value for this handle is retrieved through a call to function *ACE_get_sse_seq_handle()*.

Return Value

This function does not return a value.

ACE_restart_sse

Prototype

```
void ACE_restart_sse
(
  sse_sequence_handle_t sequence
)
```

Description

The *ACE_restart_sse()* function restarts the loop part of the sampling sequence loaded into the ACE's SSE. The sampling sequence will be restarted from the beginning of the sequence but omitting the initialization phase of the sequence. This function would typically be called after stopping the sampling sequence using the *ACE_stop_see()* function or with non-repeating sequences.

Parameters

sequence

The *sequence* parameter is the SSE sequence handle identifying the sampling sequence to load into the ACE SSE. The value for this handle is retrieved through a call to function *ACE_get_sse_seq_handle()*.

Return Value

This function does not return a value.

ACE_stop_sse

Prototype

```
void ACE_stop_sse
(
  sse_sequence_handle_t sequence
)
```

Description

The *ACE_stop_sse()* function stops execution of the SSE sequence indentified by the sequence handle passed as parameter.

Parameters

Sequence

The *sequence* parameter is the SSE sequence handle identifying the sampling sequence to load into the ACE SSE. The value for this handle is retrieved through a call to function *ACE_get_sse_seq_handle()*.

Return Value

This function does not return a value.

ACE_resume_sse

Prototype

```
void ACE_resume_sse
(
  sse_sequence_handle_t sequence
)
```

Description

The *ACE_resume_sse()* function causes the SSE sampling sequence identified by the sequence handle passed as parameter to resume execution. This function is typically used to restart execution of a sequence at the point where it was stopped through a call to *ACE_stop_sse()*.

Parameters

sequence

The *sequence* parameter is the SSE sequence handle identifying the sampling sequence to load into the ACE SSE. The value for this handle is retrieved through a call to function *ACE_get_sse_seq_handle()*.

Return Value

This function does not return a value.

Functions − Sample Sequencing Engine Interrupts Control

The following functions are used to control interrupts generated from the ACE's SSE. These interrupts would typically be used to detect when valid data is available from the ADCs controlled by the SSE or to detect the complete or partial completion of the sampling sequence through the insertion of SSE program counter general purpose interrupt assertion as part of the sequence.

- *ACE_enable_sse_irq()*
- *ACE_disable_sse_irq()*
- *ACE_clear_sse_irq()*

ACE_enable_sse_irq

Prototype

```
void ACE_enable_sse_irq
(
  sse_irq_id_t sse_irq_id
)
```

Description

The *ACE_enable_sse_irq()* function enables the Sampling Sequencing Engine (SSE) interrupt source specified as parameter to generate interrupts.

Parameters

sse_irq_id

The *sse_irq_id* parameter identifies the SSE interrupt source controlled by this function.

Return Value

This function does not return a value.

ACE_disable_sse_irq

Prototype

```
void ACE_disable_sse_irq
(
  sse_irq_id_t sse_irq_id
)
```

Description

The *ACE_disable_sse_irq()* function disables the SSE interrupt source specified as parameter from generating interrupts.

Parameters

sse_irq_id

The *sse_irq_id* parameter identifies the SSE interrupt source controlled by this function.

Return Value

This function does not return a value.

ACE_clear_sse_irq

Prototype

```
void ACE_clear_sse_irq
(
  sse_irq_id_t sse_irq_id
)
```

Description

The *ACE_clear_sse_irq()* function clears the SSE interrupt specified as parameter.

Parameters

sse_irq_id

The *sse_irq_id* parameter identifies the SSE interrupt source controlled by this function.

Return Value

This function does not return a value.

Functions − Comparators Control

The following functions are used to control the analog comparators included in the SmartFusion analog front-end. The comparator configuration functions can be used to directly configure the comparators. Their use is only required when the ACE is not configured using the ACE configurator software tool. The comparator interrupt control functions are used regardless of the way the ACE was configured to enable, disable and clear interrupts generated when the positive input of a comparator rises above or falls below the negative input.

- *ACE_set_comp_reference()*
- *ACE_set_comp_hysteresis()*
- *ACE_enable_comp()*
- *ACE_disable_comp()*
- *ACE_enable_comp_rise_irq()*
- *ACE_disable_comp_rise_irq()*
- *ACE_clear_comp_rise_irq()*
- *ACE_enable_comp_fall_irq()*
- *ACE_disable_comp_fall_irq()*
- *ACE_clear_comp_fall_irq()*
- *ACE_get_comp_status()*

ACE_set_comp_reference

Prototype

```
void ACE_set_comp_reference
(
  comparator_id_t comp_id,
  comp_reference_t reference
)
```

Description

The *ACE_set_comp_reference()* function is used to select the reference input of a temperature monitor block comparator. The reference input of a temperature monitor can be an ADC direct input or one of the SDD's outputs.

Parameters

comp_id

The *comp_id* parameter specifies the comparator for which to select the reference input. Since only temperature monitor block comparators have a selectable reference input, allowed values are:

- CMP1
- CMP3
- CMP5
- CMP7
- CMP9
- CMP11

reference

The *reference* parameter specify the signal that will be used as reference by the comparator. Allowed values are:

- SDD0_COMP_REF
- SDD1_COMP_REF
- SDD2_COMP_REF
- ADC_IN_COMP_REF

Return Value

This function does not return a value.

ACE_set_comp_hysteresis

Prototype

```
void ACE_set_comp_hysteresis
(
  comparator_id_t comp_id,
  comp_hysteresis_t hysteresis
)
```

Description

The *ACE_set_comp_hysteresis()* function is used to set the hysteresis of a comparator. There are four possible hysteresis settings: no hysteresis, +/-10mV, +/-30mV or +/-100mV.

Parameters

comp_id

The *comp_id* parameter specifies the comparator for which this function will set the hysteresis.

hysteresis

The *hysteresis* parameter specifies the hysteresis that will be applied to the comparator's input. Allowed values are:

- NO_HYSTERESIS
- HYSTERESIS_10_MV
- HYSTERESIS_30_MV
- HYSTERESIS_100_MV

Return Value

This function does not return a value.

ACE_enable_comp

Prototype

```
void ACE_enable_comp
(
  comparator_id_t comp_id
)
```

Description

The *ACE_enable_comp()* function is used to enable the comparator specified as parameter.

Parameters

comp_id

The *comp_id* parameter specifies which comparator will be enabled by a call to this function.

Return Value

This function does not return a value.

ACE_disable_comp

Prototype

```
void ACE_disable_comp
(
  comparator_id_t comp_id
)
```

Description

The *ACE_disable_comp()* function is used to disable the comparator specified as parameter.

Parameters

comp_id

The *comp_id* parameter specifies which comparator will be disabled by a call to this function.

Return Value

This function does not return a value.

ACE_enable_comp_rise_irq

Prototype

```
void ACE_enable_comp_rise_irq
(
  comparator_id_t comp_id
)
```

Description

The *ACE_enable_comp_rise_irq()* function is used to enable interrupts to be generated when the positive input of the comparator specified as parameter rises above the negative input of the comparator. The function prototypes for the comparator rise interrupt service routines are:

- *void ACE_Comp0_Rise_IRQHandler(void);*
- *void ACE_Comp1_Rise_IRQHandler(void);*
- *void ACE_Comp2_Rise_IRQHandler(void);*
- *void ACE_Comp3_Rise_IRQHandler(void);*
- *void ACE_Comp4_Rise_IRQHandler(void);*
- *void ACE_Comp5_Rise_IRQHandler(void);*
- *void ACE_Comp6_Rise_IRQHandler(void);*
- *void ACE_Comp7_Rise_IRQHandler(void);*
- *void ACE_Comp8_Rise_IRQHandler(void);*
- *void ACE_Comp9_Rise_IRQHandler(void);*
- *void ACE_Comp10_Rise_IRQHandler(void);*
- *void ACE_Comp11_Rise_IRQHandler(void);*

Parameters

comp_id

The *comp_id* parameter specifies which comparator will be enabled to generate rising interrupts.

Return Value

This function does not return a value.

ACE_disable_comp_rise_irq

Prototype

```
void ACE_disable_comp_rise_irq
(
  comparator_id_t comp_id
)
```

Description

The *ACE_disable_comp_rise_irq()* function is used to disable interrupts from being generated when the positive input of the comparator specified as parameter rises above the negative input of the comparator.

Parameters

comp_id

The *comp_id* parameter specifies which comparator will be disabled from generating rising interrupts.

Return Value

This function does not return a value.

ACE_clear_comp_rise_irq

Prototype

```
void ACE_clear_comp_rise_irq
(
  comparator_id_t comp_id
)
```

Description

The *ACE_clear_comp_rise_irq()* function is used to clear rise interrupts. This function is typically called as part of the rise interrupt service routine.

Parameters

comp_id

The *comp_id* parameter specifies the comparator for which to clear the rise interrupt.

Example

```
void ACE_Comp1_Rise_IRQHandler( void )
{
    process_rise_irq();
    ACE_clear_comp_rise_irq( CMP1 );
    NVIC_ClearPendingIRQ( ACE_Comp1_Rise_IRQn );
}
```

Return Value

This function does not return a value.

ACE_enable_comp_fall_irq

Prototype

```
void ACE_enable_comp_fall_irq
(
  comparator_id_t comp_id
)
```

Description

The *ACE_enable_comp_fall_irq()* function is used to enable interrupts to be generated when the positive input of the comparator specified as parameter falls below the negative input of the comparator. The function prototypes for the comparator fall interrupt service routines are:

- *void ACE_Comp0_Fall_IRQHandler(void);*
- *void ACE_Comp1_Fall_IRQHandler(void);*
- *void ACE_Comp2_Fall_IRQHandler(void);*
- *void ACE_Comp3_Fall_IRQHandler(void);*
- *void ACE_Comp4_Fall_IRQHandler(void);*
- *void ACE_Comp5_Fall_IRQHandler(void);*
- *void ACE_Comp6_Fall_IRQHandler(void);*
- *void ACE_Comp7_Fall_IRQHandler(void);*
- *void ACE_Comp8_Fall_IRQHandler(void);*
- *void ACE_Comp9_Fall_IRQHandler(void);*
- *void ACE_Comp10_Fall_IRQHandler(void);*
- *void ACE_Comp11_Fall_IRQHandler(void);*

Parameters

comp_id

The *comp_id* parameter specifies which comparator will be enabled to generate fall interrupts.

Return Value

This function does not return a value.

ACE_disable_comp_fall_irq

Prototype

```
void ACE_disable_comp_fall_irq
(
  comparator_id_t comp_id
)
```

Description

The *ACE_disable_comp_fall_irq()* function is used to disable interrupts from being generated when the positive input of the comparator specified as parameter falls below the negative input of the comparator.

Parameters

comp_id

The *comp_id* parameter specifies which comparator will be disabled from generating fall interrupts.

Return Value

This function does not return a value.

ACE_clear_comp_fall_irq

Prototype

```
void ACE_clear_comp_fall_irq
(
  comparator_id_t comp_id
)
```

Description

The *ACE_clear_comp_fall_irq()* function is used to clear fall interrupts. This function is typically called as part of the fall interrupt service routine.

Parameters

comp_id

The *comp_id* parameter specifies the comparator for which to clear the fall interrupt.

Return Value

This function does not return a value.

Example

```
void ACE_Comp1_Fall_IRQHandler( void )
{
    process_fall_irq();
    ACE_clear_comp_fall_irq( CMP1 );
    NVIC_ClearPendingIRQ( ACE_Comp1_Fall_IRQn );
}
```

ACE_get_comp_status

Prototype

```
uint32_t ACE_get_comp_status (void)
```

Description

The *ACE_get_comp_status()* function returns the current comparator interrupt status. It returns a 32 bit value indicating which comparators experienced a fall and/or rise event. These status bits can be cleared using the *ACE_clear_comp_rise_irq()* and *ACE_clear_comp_fall_irq()* functions.

Parameters

This function takes no parameters.

Return Value

The return value is a 32 bit number where bits 0 to 11 indicate which comparator experienced a fall event and bits 12 to 23 indicate which comparator experienced a rise event.

Functions − Sigma Delta DACs Control

The following functions are used to control the sigma delta DACs included within the SmartFusion analog front-end.

- *ACE_configure_sdd()*
- *ACE_enable_sdd()*
- *ACE_disable_sdd()*
- *ACE_set_sdd_value()*
- *ACE_set_sdd_value_sync()*

ACE_configure_sdd

Prototype

```
void ACE_configure_sdd
(
  sdd_id_t sdd_id,
  sdd_resolution_t resolution,
  uint8_t mode,
  sdd_update_method_t sync_update
)
```

Description

The *ACE_configure_sdd()* function is used to configure the operating mode of the SDD specified as parameter. It allows selecting whether the SDD will output a voltage or a current. A current between 0 and 256 µA is generated in current mode. A voltage between 0 and 2.56 V is generated in voltage mode.

This function also allows selecting whether Return-To-Zero (RTZ) mode is enabled or not. Enabling RTZ mode improves linearity of the SDD output at the detriment of accuracy. This mode should be used if linearity is more important than accuracy.

A call to this function is not required if relying on the configuration selected in the ACE configurator being loaded after reset by the system boot.

Parameters

sdd_id

The *sdd_id* parameter specifies which SDD is configured by this function. Allowed values are:

- SDD0_OUT
- SDD1_OUT
- SDD2_OUT

resolution

The *resolution* parameter specifies the desired resolution of the SDD. Allowed values are:

- SDD_8_BITS
- SDD_16_BITS
- SDD_24_BITS

mode

The *mode* parameter specifies the operating mode of the SDD. It specifies whether a current or voltage should be generated and whether RTZ mode should be used. It is a logical OR of the following defines:

- SDD_CURRENT_MODE
- SDD_VOLTAGE_MODE
- SDD_RETURN_TO_ZERO
- SDD_NON_RTZ

sync_update

The *sync_update* parameter specifies whether the SDD output will be updated individually though a call to *ACE_set_sdd_value()* or synchronously with one or more other SDD outputs via a call to *ACE_set_sdd_value_sync()*.

Return Value

This function does not return a value.

Example

```
ACE_configure_sdd
(
    SDD1_OUT,
    SDD_24_BITS,
    SDD_VOLTAGE_MODE | SDD_RETURN_TO_ZERO,
    INDIVIDUAL_UPDATE
);
```

ACE_enable_sdd

Prototype

```
void ACE_enable_sdd
(
  sdd_id_t sdd_id
)
```

Description

The *ACE_enable_sdd()* function is used to enable a Sigma Delta DAC.

Parameters

sdd_id

The *sdd_id* parameter specifies the Sigma Delta DAC to enable.

Return Value

This function does not return a value.

ACE_disable_sdd

Prototype

```
void ACE_disable_sdd
(
  sdd_id_t sdd_id
)
```

Description

The *ACE_disable_sdd()* function is used to disable a Sigma Delta DAC.

Parameters

sdd_id

The *sdd_id* parameter specifies the Sigma Delta DAC to disable.

Return Value

This function does not return a value.

ACE_set_sdd_value

Prototype

```
void ACE_set_sdd_value
(
  sdd_id_t sdd_id,
  uint32_t sdd_value
)
```

Description

The *ACE_set_sdd_value()* function is used to set the value of the output of a Sigma Delta DAC. It uses the ACE's phase accumulator to generate the one bit input bit stream into the SDD which will in turn define the voltage or current generated at the SDD output.

The SDD output is proportional to the sdd_value passed to this function taking the SDD resolution into account. A maximum voltage of 2.56V or a maximum current of 256µA will be generated when the sdd_value is set the maximum value allowed by the SDD resolution

Parameters

sdd_id

The *sdd_id* parameter specifies which Sigma Delta DAC is being set.

sdd_value

The *sdd_value* parameter specifies the value of the Sigma Delta DAC output. It is a fraction of SDD resolution. The voltage/current value generated from the sdd_value paramenter can be determined using the following equation:

sdd_output = (sdd_value / sdd_resolution) * sdd_range

where sdd_resolution is the resolution of the SDD as set through function *ACE_configure_sdd()* and sdd_range is either the maximum voltage range (2.56V) or maximum current (256µA) depending on the SDD operating mode.

Return Value

This function does not return a value.

ACE_set_sdd_value_sync

Prototype

```
void ACE_set_sdd_value_sync
(
  uint32_t sdd0_value,
  uint32_t sdd1_value,
  uint32_t sdd2_value
)
```

Description

The *ACE_set_sdd_value_sync()* function is used to synchronize the update of multiple SDD outputs.

Parameters

sdd0_value

The *sdd0_value* parameter specifies the value that will be used to set the output of SDD0. The define SDD_NO_UPDATE can be used to specify that the output value of SDD0 should not be modified.

sdd1_value

The *sdd1_value* parameter specifies the value that will be used to set the output of SDD1. The define SDD_NO_UPDATE can be used to specify that the output value of SDD1 should not be modified.

sdd2_value

The *sdd2_value* parameter specifies the value that will be used to set the output of SDD2. The define SDD_NO_UPDATE can be used to specify that the output value of SDD2 should not be modified.

Return Value

This function does not return a value.

Example

For example the code below will change the output value of SDD0 and SDD2 so that the voltage/current generated by SDD0 and ADD2 will change at the same time. This function call will not affect the output value of SDD1.

```
uint32_t sdd0_value = 0x1234;
uint32_t sdd2_value = 0x5678;
ACE_set_sdd_value_sync( sdd0_value, SDD_NO_UPDATE, sdd2_value );
```

Product Support

Actel backs its products with various support services including Customer Service, a Customer Technical Support Center, a web site, an FTP site, electronic mail, and worldwide sales offices. This appendix contains information about contacting Actel and using these support services.

Customer Service

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

From Northeast and North Central U.S.A., call **650.318.4480**
From Southeast and Southwest U.S.A., call **650. 318.4480**
From South Central U.S.A., call **650.318.4434**
From Northwest U.S.A., call **650.318.4434**
From Canada, call **650.318.4480**
From Europe, call **650.318.4252** or **+44 (0) 1276 401 500**
From Japan, call **650.318.4743**
From the rest of the world, call **650.318.4743**
Fax, from anywhere in the world **650. 318.8044**

Actel Customer Technical Support Center

Actel staffs its Customer Technical Support Center with highly skilled engineers who can help answer your hardware, software, and design questions. The Customer Technical Support Center spends a great deal of time creating application notes and answers to FAQs. So, before you contact us, please visit our online resources. It is very likely we have already answered your questions.

Actel Technical Support

Visit the Actel Customer Support website (http://www.actel.com/support/search/default.aspx) for more information and support. Many answers available on the searchable web resource include diagrams, illustrations, and links to other resources on the Actel web site.

Website

You can browse a variety of technical and non-technical information on Actel's home page, at http://www.actel.com/.

Contacting the Customer Technical Support Center

Highly skilled engineers staff the Technical Support Center from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. Several ways of contacting the Center follow:

Email

You can communicate your technical questions to our email address and receive answers back by email, fax, or phone. Also, if you have design problems, you can email your design files to receive assistance. We constantly monitor the email account throughout the day. When sending your request to us, please be sure to include your full name, company name, and your contact information for efficient processing of your request.

The technical support email address is tech@actel.com.

Phone

Our Technical Support Center answers all calls. The center retrieves information, such as your name, company name, phone number and your question, and then issues a case number. The Center then forwards the information to a queue where the first available application engineer receives the data and returns your call. The phone hours are from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. The Technical Support numbers are:

650.318.4460
800.262.1060

Customers needing assistance outside the US time zones can either contact technical support via email (tech@actel.com) or contact a local sales office. Sales office listings can be found at www.actel.com/company/contact/default.aspx.

Actel is the leader in low-power FPGAs and mixed-signal FPGAs and offers the most comprehensive portfolio of system and power management solutions. Power Matters. Learn more at http://www.actel.com .

Actel Corporation *• 2061 Stierlin Court • Mountain View, CA 94043 • USA*
Phone 650.318.4200 • Fax 650.318.4600 • Customer Service: 650.318.1010 • Customer Applications Center: 800.262.1060

Actel Europe Ltd*. • River Court, Meadows Business Park • Station Approach, Blackwater • Camberley Surrey GU17 9AB • United Kingdom*
Phone +44 (0) 1276 609 300 • Fax +44 (0) 1276 607 540

Actel Japan *• EXOS Ebisu Building 4F • 1-24-14 Ebisu Shibuya-ku • Tokyo 150 • Japan*
Phone +81.03.3445.7671 • Fax +81.03.3445.7668 • http://jp.actel.com

Actel Hong Kong *• Room 2107, China Resources Building • 26 Harbour Road • Wanchai • Hong Kong*
Phone +852 2185 6460 • Fax +852 2185 6488 • www.actel.com.cn

50200196-2/6.10