ANALOG DEVICES

DEVICE ACCESS SERVICE

PLATFORM TOOLS GROUP

DATE: OCTOBER 12, 2007

Table of Contents

1. Device Access Service Overview 5

2. Files 6

2.1. Include Files 6

2.2. Source Files 6

3. Theory of Operation 7

3.1. Device Access Commands 7

3.1.1. Single register access 7

3.1.2. Specific register field access 7

3.1.3. Register table access 8

3.1.4. Register field table access 8

3.1.5. Register block access 9

3.2. Function – adi_device_access( ) 10

3.3. Device access service specific return codes 15

4. Examples 16

4.1. Device Driver header file 17

4.2. Calling Device Access Service 18

4.3. Device access command examples 22

4.3.1. Single register access 22

4.3.2. Specific register field access 22

4.3.3. Register table access 23

4.3.4. Register Field table access 23

4.3.5. Register block access 24

Table of Figures

Table 1 – Revision History 4
Table 2 – Commands to access a single register 7
Table 3 – Commands to access a specific register field 7
Table 4 – Commands to access table of registers 8
Table 5 – Commands to access table of register(s) fields 8
Table 6 – Commands to access consecutive block of registers 9
Table 7 – Device Access Callback Function arguments 11

Document Revision History

Date	Description of Changes
Feb 07, 2006	Initial Release
Mar 14, 2006	Added example section for device access service commands
May 15, 2006	Updated to new device access interface
Sep 11, 2006	Added TWI/SPI Device (global) address info
Oct 12, 2007	Added support to SPI based devices with R/W flag at bit 7 or bit 15

Table 1 – Revision History

1. Device Access Service Overview

The Device Access Service provides a simple, effective and easy way to read/write internal registers of devices that supports TWI or SPI access. Programmer can access the device register(s) or register filed(s) by issuing a relevant command among the ten unique commands and these commands should be accompanied with a specific value that holds needed information to perform register access. The service will return a code indicating success/failure of register access.

2. Files

The files listed below comprise the Device Access Service API and source files.

2.1. Include Files

The driver sources include the following include files:

- **<services/services.h>**
  This file contains all definitions, function prototypes etc. for all the System Services.

- **<drivers/adi_dev.h>**
  This file contains all definitions, function prototypes etc. for the Device Manager and general device driver information.

- **<drivers/deviceaccess/adi_device_access.h>**
  This file contains all definitions, function prototypes etc. specific to TWI or SPI Device Access.

2.2. Source Files

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

- **<Blackfin/lib/src/drivers/deviceaccess/adi_device_access.c>**
  This file contains all the source code for the TWI and SPI based Device Access Service. All source code is written in 'C'. There are no assembly level functions in this driver.

- **<Blackfin/lib/src/drivers/deviceaccess/adi_device_access_spi.c>**
  This file defines a macro that directs the compiler to build SPI based device access service.

- **<Blackfin/lib/src/drivers/deviceaccess/adi_device_access_twi.c>**
  This file defines a macro that directs the compiler to build TWI based device access service.

3. Theory of Operation

3.1. Device Access Commands

The device access service supports following ten commands listed in device manager header file (adi_dev.h).

Device access hierarchy:

1. Application program calls device manager function *adi_dev_control( )* with one of the device access commands accompanied with device identifier handle and command specific value.
2. Device manager calls corresponding device driver function *adi_pdd_control( )* with the command id & corresponding command value passed by the application.
3. Device driver calls device access function *adi_device_access( )* with a pointer to device access data structure of type *ADI_DEVICE_ACCESS_REGISTERS* (refer section 1.1).
4. Device Access service reads/writes from/to the list of registers and returns result code (refer section 3.3)

3.1.1. Single register access

The following commands can read from / write to a single device register.

Command	Description
ADI_DEV_CMD_REGISTER_READ	Reads a single device register
ADI_DEV_CMD_REGISTER_WRITE	Writes to a single device register

Table 2 – Commands to access a single register

Command specific value: Pointer to a data structure of type ADI_DEV_ACCESS_REGISTER

```
typedef struct ADI_DEV_ACCESS_REGISTER
{
        u16     Address;        // Device register address
        u16     Data;           // Data read/written from/to the register
} ADI_DEV_ACCESS_REGISTER;
```

Refer to section 4.3.1 for single register access examples

3.1.2. Specific register field access

The following commands can read from / write to a specific field location in a single device register.

Command	Description
ADI_DEV_CMD_REGISTER_FIELD_READ	Reads a specific field location in a single device register.
ADI_DEV_CMD_REGISTER_FIELD_WRITE	Writes to a specific field location in a single device register

Table 3 – Commands to access a specific register field

Command specific value: Pointer to a data structure of type ADI_DEV_ACCESS_REGISTER_FIELD

```
typedef struct ADI_DEV_ACCESS_REGISTER_FIELD
{
        u16 Address;        // Register address to access
        u16 Field;          // Register field in the above address to be accessed
        u16 Data;           // Register field data read/written from/to the device
} ADI_DEV_ACCESS_REGISTER_FIELD;
```

Refer to corresponding device driver header for the list of accessible register fields. Refer to section 4.3.2 for specific register field access examples

3.1.3. Register table access

The following commands can read from / write to a table of selective device registers. These registers need not be consecutive and the register table **MUST** be terminated with a device access delimiter (ADI_DEV_REGEND).

Command	Description
ADI_DEV_CMD_REGISTER_TABLE_READ	Reads a table of selective device registers
ADI_DEV_CMD_REGISTER_TABLE_WRITE	Writes to a table of selective device registers

Table 4 – Commands to access table of registers

Command specific value: Pointer to an array of data structure of type ADI_DEV_ACCESS_REGISTER

```
typedef struct ADI_DEV_ACCESS_REGISTER
{
        u16     Address;    // Device register address
        u16     Data;       // Data read/written from/to the register
} ADI_DEV_ACCESS_REGISTER;
```

Refer to section 4.3.3 for selective register table access examples

3.1.4. Register field table access

The following commands can read from / write to a table of selective device register(s) fields. These registers / field locations need not be consecutive and the register field table **MUST** be terminated with a device access delimiter (ADI_DEV_REGEND).

Command	Description
ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ	Reads a table of selective device register fields
ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE	Writes to a table of selective device register fields

Table 5 – Commands to access table of register(s) fields

Command specific value: Pointer to an array of data structure of type ADI_DEV_ACCESS_REGISTER_FIELD

```
typedef struct ADI_DEV_ACCESS_REGISTER_FIELD
{
        u16 Address;        // Register address to access
        u16 Field;          // Register field in the above address to be accessed
        u16 Data;           // Register field data read/written from/to the device
} ADI_DEV_ACCESS_REGISTER_FIELD;
```

Refer to corresponding device driver header file for list of accessible register fields. Refer to section 4.3.4 for selective register field table access examples

3.1.5. Register block access

The following commands can access ‘Count’ number of consecutively located registers in a device starting with the given ‘register block’ start address.

Command	Description
ADI_DEV_CMD_REGISTER_BLOCK_READ	Reads a block of consecutive device registers
ADI_DEV_CMD_REGISTER_BLOCK_WRITE	Writes to a block of consecutive device registers

Table 6 – Commands to access consecutive block of registers

Command specific value: Pointer to a data structure of type ADI_DEV_ACCESS_REGISTER_BLOCK

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

Refer to section 4.3.5 for register block access examples

3.2. Function – adi_device_access( )

Any device driver or application aim to use this service should call the device access function – *adi_device_access (ADI_DEVICE_ACCESS_REGISTERS *Device)*.

Refer to section 4.2 for *adi_device_access( )* function example

Prototype

u32 adi_device_access (ADI_DEVICE_ACCESS_REGISTERS *Device);

Arguments

*Device – pointer to a structure of type ADI_DEVICE_ACCESS_REGISTERS

ADI_DEVICE_ACCESS_REGISTERS is a structure holding device specific information

```
typedef struct ADI_DEVICE_ACCESS_REGISTERS
{
    ADI_DEV_MANAGER_HANDLE              ManagerHandle;
    void                                *ClientHandle;
    u32                                 DeviceNumber;
    u32                                 DeviceAddress;
    ADI_DCB_HANDLE                      DCBHandle;
    ADI_DEVICE_ACCESS_CALLBACK_FN       DeviceFunction;
    u32                                 Command;
    void                                *Value;
    u32                                 FinalRegAddr;
    ADI_DEVICE_ACCESS_REGISTER_FIELD    *RegisterField;
    ADI_DEVICE_ACCESS_RESERVED_VALUES   *ReservedValues;
    ADI_DEVICE_ACCESS_VALIDATE_REGISTER *ValidateRegister;
    ADI_DEV_CMD_VALUE_PAIR              *ConfigTable;
    ADI_DEVICE_ACCESS_SELECT            *SelectAccess;
    void                                *pAdditionalinfo;
}  ADI_DEVICE_ACCESS_REGISTERS;
```

ManagerHandle – Handle to the Device Manager

***ClientHandle**

This is an identifier defined by the device driver calling this service. The Device Access Service passes this value back to the device driver as an argument in the Device Access Callback Function.

DeviceNumber – TWI or SPI Device number to be used to access the device registers

DeviceAddress – TWI address or SPI Global address of the device to be accessed

A valid TWI device address **MUST** be provided to access internal register(s) of any TWI based device. Any slave device that communicates with Blackfin via TWI will/must have a valid device address, typical format would be a 7-bit device address + one R/W bit. Please note that only the 7-bit device address value should be loaded to the above field. Analog Devices TWI driver will append the required R/W bit while initiating communication with the corresponding device.

SPI based devices may or may not have a Global address and devices with no such global address can be accessed / controlled via Blackfin SPI chip select pins.

DCBHanble – Handle to the Deferred Callback Service that is used for the device calling this service

DeviceFunction

This is the Device Access Callback Function specific to the driver. The device access service will call this function before accessing any of the device registers.

Device Access Callback Function prototype

u32 (*ADI_DEVICE_ACCESS_CALLBACK_FN) (void*,u16*,u16,ADI_DEVICE_ACCESS_MODE);

Arguments:

void*	Identifier passed by the device driver calling this service (ClientHandle)
u16*	Pointer to the location holding the address of next device register to be accessed
u16	Value of the next device register to be accessed. The value is valid only when the device is in PRE_WRITE mode.
ADI_DEVICE_ACCESS_MODE	This indicates the present access mode of the device. The device can be in one of the following modes ADI_DEVICE_ACCESS_PRE_READ – The register address passed to this function is the next to be read. ADI_DEVICE_ACCESS_PRE_WRITE – The register address passed to this function is the next to be configured. ADI_DEVICE_ACCESS_PRE_ADDR_GENERATE – The register address passed to this function will be used to generate the next device register address to be accessed. This mode is passed only when the device registers are accessed in blocks.

Table 7 – Device Access Callback Function arguments

Note:

- In ***register block access*** operation, the device access service calls the driver specific function *(DeviceFunction)* with following parameters before generating the next register address to be accessed.

  Parameters – Client Handle, Address of previously accessed register, value of previously accessed register, and Device Access Mode as ADI_DEVICE_ACCESS_PRE_ADDR_GENERATE. If the device driver intend to change the next register address to be accessed (example devices with two or more register banks), the device driver should load the register address with the value one less than the next register address to be accessed next. The driver should also update the ADI_DEVICE_ACCESS_REGISTERS structure as it will be affected by register bank change. This can be achieved by passing address of ADI_DEVICE_ACCESS_REGISTERS structure as a Client Handle. Refer to AD7183 device driver for further information.

Command

This is the command identifier. Refer to section 3.1 for list of commands supported by this service.

***Value**

This is the address of command–specific parameter. Refer to section 3.1 for list of specific values to be passed with the device access commands

Data to/from a TWI operated device is sent to/read from the TWI driver with out any modifications.

For any SPI operated device with 8 or 16 bit register address, the device access service appends a read/write bit at the end of first byte of each data packet sent to the device. This bit indicates the device hardware to read or write the specified register. In case of SPI driven devices with uneven register address or register data length (example AD1836A), no such data modifications will be done.

FinalRegAddr – Final register address of the device to be accessed

***RegisterField**

This holds the address of a structure type ADI_DEVICE_ACCESS_REGISTER_FIELD and is used in selective register field access and to validate the selected register field of the device to be accessed (Refer to page 18 for more information)

```
typedef struct ADI_DEVICE_ACCESS_REGISTER_FIELD
{
    u32         Count;
    u16         *RegAddr;
    u16         *RegField;
}   ADI_DEVICE_ACCESS_REGISTER_FIELD;
```

Count	'Count' of device registers containing individual fields
*RegAddr	Pointer to an array of device register addresses containing individual fields
*RegField	Pointer to an array of device register field locations in the corresponding registers

***ReservedValues**

This holds the address of a structure type ADI_DEVICE_ACCESS_RESERVED_VALUES and is used to configure reserved bit locations in the device to its recommended value (Refer to page 19 for more information)

```
typedef struct ADI_DEVICE_ACCESS_RESERVED_VALUES
{
    u32         Count;
    u16         *RegAddr;
    u16         *ReservedBits;
    u16         *ReservedBitValue;
}   ADI_DEVICE_ACCESS_RESERVED_VALUES;
```

Count	'Count' of device registers containing reserved bit locations
*RegAddr	Pointer to an array of device register addresses containing reserved bit locations
*ReservedBits	Pointer to an array of reserved bit locations in corresponding device registers
*ReservedBitValue	Pointer to an array of recommended values to the corresponding reserved bit locations

***ValidateRegiste**r

This holds the address of a structure type ADI_DEVICE_ACCESS_VALIDATE_REGISTER and is used to validate the device register address to be accessed (Refer to page 19 for more information).

```
typedef struct ADI_DEVICE_ACCESS_VALIDATE_REGISTER
{
    u32         Count1;
    u16         *InvalidRegs;
    u32         Count2;
    u16         *ReadOnlyRegs;
}   ADI_DEVICE_ACCESS_VALIDATE_REGISTER;
```

Count1	'Count' of Invalid register addresses in a device
*InvalidRegs	Pointer to an array of invalid register addresses in a device
Count2	'Count' of Read-only register addresses in a device
*ReadOnlyRegs	Pointer to an array of read-only registers in a device

***ConfigTable**

This holds the address of a TWI or SPI device configuration table. To access any TWI based device, the driver/application **MUST** pass a TWI configuration table to this service. The configuration table is optional for any SPI based device, as the device access service sets the Blackfin SPI in following mode by default.

- SPI Dataflow Method as ADI_DEV_MODE_CHAINED
- SPI Baud Register value as 0x7FF
- Sets Blackfin SPI in Master mode
- Sets clock polarity bit (active low SCLK)
- Sets clock phase bit (SPI software chip-select)
- SPI word length as 8 (cannot be altered)

***SelectAccess**

This holds the address of a structure type ADI_DEVICE_ACCESS_SELECT and is used categorise the device to be accessed (Refer to page 20 for more information).

```
typedef struct ADI_DEVICE_ACCESS_SELECT
{
    u8                                  DeviceCS;
    ADI_DEVICE_ACCESS_SET_DEVICE_TYPE   Gaddr_len;
    ADI_DEVICE_ACCESS_SET_DEVICE_TYPE   Raddr_len;
    ADI_DEVICE_ACCESS_SET_DEVICE_TYPE   Rdata_len;
    ADI_DEVICE_ACCESS_SET_ACCESS_TYPE   AccessType;
}   ADI_DEVICE_ACCESS_SELECT;
```

DeviceCS	Device SPI Chip-select value (not valid for TWI operated devices)
Gaddr_len	SPI Global address length of the device to be accessed (not valid for TWI operated devices).
Raddr_len	Register address length of the device to be accessed.
Rdata_len	Register data length of the device to be accessed
AccessType	Sets access type of the selected device (TWI or SPI)

ADI_DEVICE_ACCESS_SET_DEVICE_TYPE is an enumerated data which can take one of the following values

ADI_DEVICE_ACCESS_SET_DEVICE_TYPE Enumeration IDs	Comments
ADI_DEVICE_ACCESS_LENGTH0	Device has no SPI/TWI (global) address / Device has no registers to access / No register value(s) available for this device
ADI_DEVICE_ACCESS_LENGTH1	Device SPI/TWI (global) address is of 1 byte length / Device register address is of 1 byte length / Device register value is of 1 byte length
ADI_DEVICE_ACCESS_LENGTH2	Device SPI/TWI (global) address is of 2 bytes length / Device register address is of 2 bytes length / Device register value is of 2 bytes length

ADI_DEVICE_ACCESS_SET_ACCESS_TYPE is an enumerated data which can take one of the following values

ADI_DEVICE_ACCESS_SET_ACCESS_TYPE Enumeration IDs	Comments
ADI_DEVICE_ACCESS_TYPE_TWI	Device supports TWI based register access
ADI_DEVICE_ACCESS_TYPE_SPI	Device supports SPI based register access and has SPI read/write flag at bit 0 of its global or register address.
ADI_DEVICE_ACCESS_TYPE_SPI_1	Device supports SPI based register access and has SPI read/write flag at bit 7 of its global or register address.
ADI_DEVICE_ACCESS_TYPE_SPI_2	Device supports SPI based register access and has SPI read/write flag at bit 15 of its global or register address.

***pAdditionalinfo**

To be used for future extensions.

Note:

- Device Access Service supports devices only with register address / register data length of up to 16 bits.
- This service does not support devices that send or receive data with LSB first.
- Avoid using Selective Field Access for write-only register as the bits other than the selected field will be marked as zero.
- Device Access Callback Function *(DeviceFunction)* can be used to keep in track of registers whose values can affect the driver functionality (refer AD1836A_ii / AD1938_ii / AD717x device drivers for more information)
- For devices with multiple banks of registers, whenever a bank switch occurs, the driver should update device access service with the new register address (for Block Access only) and new RegisterField / ReservedValues / ValidateRegister table address corresponding to the selected bank. These operations can be carried out in the Device Access Callback Function *(DeviceFunction)*. Refer to ADV7183 device driver for more information.

3.3. Device access service specific return codes

- ADI_DEV_RESULT_TWI_LOCKED
  - Indicates the present TWI device is locked in other operation
- ADI_DEV_RESULT_REQUIRES_TWI_CONFIG_TABLE
  - Client need to supply a configuration table for the TWI driver
- ADI_DEV_RESULT_CMD_NOT_SUPPORTED
  - Command not supported by the Device Access Service
- ADI_DEV_RESULT_INVALID_REG_ADDRESS
  - The client attempting to access an invalid register address
- ADI_DEV_RESULT_INVALID_REG_FIELD
  - The client attempting to access an invalid register field location
- ADI_DEV_RESULT_INVALID_REG_FIELD_DATA
  - The client attempting to write an invalid data to selected register field location
- ADI_DEV_RESULT_ATTEMPT_TO_WRITE_READONLY_REG
  - The client attempting to write to a read-only location
- ADI_DEV_RESULT_ATTEMPT_TO_ACCESS_RESERVE_AREA
  - The client attempting to access a reserved location
- ADI_DEV_RESULT_ACCESS_TYPE_NOT_SUPPORTED
  - Device Access Service does not support the access type provided by the driver

4. Examples

Examples showing how to use the Device Access Service to write a driver for TWI or SPI operated device.

Consider a device (ADxxxx) with six internal registers. The register address is eight bits length and each register is eight bits wide. Bit locations marked as X are reserved bits and should be set to the value indicated within braces adjacent to it.

Register definitions:

DevReg1 (Address – 0x02) (Read and Write)

Bits	7	6	5	4	3	2	1	0
Field	Field4	X(1)	Field3			Field2		Field1

DevReg2 (Address – 0x03) (Read and Write)

Bits	7	6	5	4	3	2	1	0
Field	Field4				Field3	Field2	Field1	

DevReg3 (Address – 0x05) (Read-only)

Bits	7	6	5	4	3	2	1	0
Field	Field6		Field5	Field4	Field3		Field2	Field1

DevReg4 (Address – 0x06) (Read and Write)

Bits	7	6	5	4	3	2	1	0
Field	Field4	X(1)	Field3	X(0)	X(1)	Field2		Field1

DevReg5 (Address – 0x08) (Read and Write)

Bits	7	6	5	4	3	2	1	0
Field	No individual field(s)							

DevReg6 (Address – 0x09) (Read and Write)

Bits	7	6	5	4	3	2	1	0
Field	X(0)	X(0)	X(1)	X(1)	X(0)	Field1		

4.1. Device Driver header file

This section explains how to define the above registers and its field in the driver header file (adi_adxxxx.h)

Define the registers with its corresponding register address as it value

```
#define ADxxxx_DEVREG1      0x02
#define ADxxxx_DEVREG2      0x03
#define ADxxxx_DEVREG3      0x05
#define ADxxxx_DEVREG4      0x06
#define ADxxxx_DEVREG5      0x08
#define ADxxxx_DEVREG6      0x09
```

Define individual fields of the register with the starting bit location of a field as its value

```
// Individual field definitions for DevReg1 (ADxxxx_DEVREG1)

#define ADxxxx_DEVREG1_FIELD1     0
#define ADxxxx_DEVREG1_FIELD2     1
#define ADxxxx_DEVREG1_FIELD3     3
#define ADxxxx_DEVREG1_FIELD4     7

// Individual field definitions for DevReg2 (ADxxxx_DEVREG2)

#define ADxxxx_DEVREG2_FIELD1     0
#define ADxxxx_DEVREG2_FIELD2     2
#define ADxxxx_DEVREG2_FIELD3     3
#define ADxxxx_DEVREG2_FIELD4     4

// Individual field definitions for DevReg3 (ADxxxx_DEVREG3)

#define ADxxxx_DEVREG3_FIELD1     0
#define ADxxxx_DEVREG3_FIELD2     1
#define ADxxxx_DEVREG3_FIELD3     2
#define ADxxxx_DEVREG3_FIELD4     4
#define ADxxxx_DEVREG3_FIELD5     5
#define ADxxxx_DEVREG3_FIELD6     6

// Individual field definitions for DevReg4 (ADxxxx_DEVREG4)

#define ADxxxx_DEVREG4_FIELD1     0
#define ADxxxx_DEVREG4_FIELD2     1
#define ADxxxx_DEVREG4_FIELD3     5
#define ADxxxx_DEVREG4_FIELD4     7

// Individual field definitions for DevReg6 (ADxxxx_DEVREG6)

#define ADxxxx_DEVREG6_FIELD1     0
```

4.2. Calling Device Access Service

This section explains how to assign values to the structure (ADI_DEVICE_ACCESS_REGISTERS) and to call the device access service.

```
// define the structure to hold the device access service values
ADI_DEVICE_ACCESS_REGISTERS        access_adxxxx;
```

ManagerHandle

```
//Load the Manager handle passed by the application program
access_adxxxx.ManagerHandle = ManagerHandle;
```

***ClientHandle**

```
// Identifier specific to the ADxxxx driver (here NULL for example)
access_adxxxx.ClientHandle = NULL;
```

DeviceNumber

```
//TWI or SPI device number to be used by the ADxxxx driver
access_adxxxx.DeviceNumber = TWI_SPI_DevNumber;
```

DeviceAddress

```
// TWI or SPI Global Address of the ADxxxx device – MUST load a valid address for TWI based device
// refer to page 10 for more  information
access_adxxxx.DeviceAddress = ADxxxxDevAddress;
```

DCBHandle

```
// Pass the deferred callback handle passed by the application program
access_adxxxx.DCBHandle = DCBHandle;
```

DeviceFunction

```
/* Function specific to ADxxxx driver. This function will be called every time when the device access service tries
to access an ADxxxx device register. */
access_adxxxx.DeviceFunction = ADxxxxAccessFn;
```

Command

```
// Command to access ADxxxx internal registers
access_adxxxx.Command = Command;
```

***Value**

```
// Value corresponding to the command issued to access ADxxxx internal registers
access_adxxxx.Value = Value;
```

FinalRegAddr

```
// ADxxxx final register address
access_adxxxx.FinalRegAddr = 0x09;
```

***RegisterField**

```
// This table holds information used to access individual register fields of ADxxxx (defined in section 4 & 4.1)
ADI_DEVICE_ACCESS_REGISTER_FIELD RegisterField;

// ADxxxx register addresses with individual register fields
u16 adxxxxRegAddr1[ ] = { ADxxxx_DEVREG1,  ADxxxx_DEVREG2,  ADxxxx_DEVREG3,
                          ADxxxx_DEVREG4, ADxxxx_DEVREG6};
```

```
// Register Field locations corresponding to the entries in adxxxxRegAddr1
// Reserved bit locations are marked as 1
// First bit of a new field is marked as 1 and remaining bits of the field are marked 0
u16 adxxxxRegField[ ] = {  0xCB, 0x1D, 0x77, 0xFB, 0xF9 };

// 'Count' of ADxxxx Register locations containing individual fields
RegisterField.Count = sizeof(adxxxxRegAddr1) / 2 ;
// Pointer to array of ADxxxx register locations containing individual fields
RegisterField.RegAddr = & adxxxxRegAddr1[0] ;
// Pointer to array of ADxxxx register field locations corresponding to entries in array adxxxxRegAddr1
RegisterField.RegField = & adxxxxRegField[0] ;

// Table to access ADxxxx individual register fields
access_adxxxx.RegisterField = & RegisterField;
```

***ReservedValues**

```
// This table holds information used to configure reserved bit locations in ADxxxx to its recommended value
ADI_DEVICE_ACCESS_RESERVED_VALUES ReservedValues;

// ADxxxx register addresses with reserved locations (defined in section 4 & 4.1)
u16 adxxxxRegAddr2[ ] = { ADxxxx_DEVREG1, ADxxxx_DEVREG4, ADxxxx_DEVREG6};

// Reserved bit locations corresponding to the entries in adxxxxRegAddr2
// Reserved bit locations are marked as 1 and others as 0
u16 adxxxxReservedBits [ ] = {  0x40, 0x58, 0xF8 };

// Recommended values for the Reserved bit locations corresponding to entries in adxxxxRegAddr2
// Reserved bit values are marked as with its recommended value and others field locations as 0
u16 adxxxxReservedValues [ ] = {  0x40, 0x48, 0x30 };

// 'Count' of ADxxxx Register locations containing Reserved bit locations
ReservedValues.Count = sizeof(adxxxxRegAddr2) / 2 ;
// Pointer to array of ADxxxx register locations containing reserved bits
ReservedValues.RegAddr = & adxxxxRegAddr2[0] ;
// Pointer to array of reserved bit locations corresponding to entries in array adxxxxRegAddr2
ReservedValues.ReservedBits = & adxxxxReservedBits[0] ;
// Pointer to array of recommended values for ADxxxx reserved bit locations
ReservedValues. ReservedBitValue = & adxxxxReservedValues[0] ;

// Table to update ADxxxx reserved bit locations
access_adxxxx.ReservedValues = & ReservedValues;
```

***ValidateRegister**

```
// This table holds information used to validate ADxxxx register address to be accessed
ADI_DEVICE_ACCESS_VALIDATE_REGISTER ValidateRegister;

// Invalid register locations in ADxxxx
u16 adxxxxInvalidRegs [ ] = { 0x00, 0x01, 0x04, 0x07 };

// Read-only register address location(s) in ADxxxx (defined in section 4 & 4.1)
u16 adxxxxReadOnlyRegs [ ] = { ADxxxx_DEVREG3 };

// 'Count' of Invalid register locations in ADxxxx
ValidateRegister.Count1 = sizeof(adxxxxInvalidRegs) / 2 ;
// Pointer to array of Invalid register locations in ADxxxx
ValidateRegister.InvalidRegs = & adxxxxInvalidRegs [0] ;
```

```
// 'Count' of Read-only location(s) in ADxxxx
ValidateRegister.Count2 = sizeof(adxxxxReadOnlyRegs) / 2 ;
// Pointer to array of Read-only location(s) in ADxxxx
ValidateRegister.ReadOnlyRegs = & adxxxxReadOnlyRegs [0] ;

// Table to validate ADxxxx register address
access_adxxxx.ValidateRegister = & ValidateRegister;
```

***ConfigTable**

```
// TWI or SPI device configuration table address specific to ADxxxx. MUST pass a valid TWI configuration table
// refer to page 13 for more information
access_adxxxx.ConfigTable = TWI_SPI_Config;
```

***SelectAccess**

```
//This table categorises ADxxxx as TWI or SPI based device
ADI_DEVICE_ACCESS_SELECT   SelectAccess ;
```

Case1: Consider ADxxxx to be TWI based device

```
// Blackfin Chip-select (Don't care for TWI)
SelectAccess.DeviceCS = 0;
// ADxxxx Global address length (Don't care for TWI)
SelectAccess.Gaddr_len = ADI_DEVICE_ACCESS_LENGTH0;
// ADxxxx access type (TWI)
SelectAccess.AccessType = ADI_DEVICE_ACCESS_TYPE_TWI;
```

Case 1a: ADxxxx is a device with 8-bit register address and 8-bit register value.

```
// ADxxxx register address length (1 byte)
SelectAccess.Raddr_len = ADI_DEVICE_ACCESS_LENGTH1;
// ADxxxx register data length (1 byte)
SelectAccess.Rdata_len = ADI_DEVICE_ACCESS_LENGTH1;
```

Case 1b: ADxxxx is a device with 16-bit register address and 16-bit register value.

```
// ADxxxx register address length (2 bytes)
SelectAccess.Raddr_len = ADI_DEVICE_ACCESS_LENGTH2;
// ADxxxx register data length (2 bytes)
SelectAccess.Rdata_len = ADI_DEVICE_ACCESS_LENGTH2;
```

Case2: Consider ADxxxx to be SPI based device with read/write flag at bit 0

```
// Blackfin SPI Chip-select for ADxxxx
SelectAccess.DeviceCS = ADxxxx_SPI_CS;
// ADxxxx access type (SPI)
SelectAccess.AccessType = ADI_DEVICE_ACCESS_TYPE_SPI;
```

Case 2a: ADxxxx is a device with No Global address, 8-bit register address and 16-bit register value.

```
// ADxxxx Global address length (0 byte)
SelectAccess.Gaddr_len = ADI_DEVICE_ACCESS_LENGTH0;
// ADxxxx register address length (1 byte)
SelectAccess.Raddr_len = ADI_DEVICE_ACCESS_LENGTH1;
// ADxxxx register data length (2 bytes)
SelectAccess.Rdata_len = ADI_DEVICE_ACCESS_LENGTH2;
```

Case 2b: ADxxxx is a device with 16-bit Global address, 16-bit register address and 16-bit register value.

```
// ADxxxx Global address length (2 bytes)
SelectAccess.Gaddr_len = ADI_DEVICE_ACCESS_LENGTH2;
// ADxxxx register address length (2 bytes)
SelectAccess.Raddr_len = ADI_DEVICE_ACCESS_LENGTH2;
// ADxxxx register data length (2 bytes)
SelectAccess.Rdata_len = ADI_DEVICE_ACCESS_LENGTH2;
```

Case 2c: ADxxxx is a special device with No Global address, 4-bit register address and 9-bit register value (example AD1836A). For such device, the device access service cannot perform operations such as adding SPI read/write bit, register field access and register address validation. Such operations should be carried out by the device driver itself where as the device access service can be used to access internal registers of the device via SPI. The 'access_adxxxx' structure should specifically be assigned with following values for such special devices. Refer AD1836A device driver for further information.

```
// ADxxxx Global address length
SelectAccess.Gaddr_len = ADI_DEVICE_ACCESS_LENGTH0;
// ADxxxx register address length (2 bytes – has both register address and register data in it)
SelectAccess.Raddr_len = ADI_DEVICE_ACCESS_LENGTH2;
// ADxxxx register data length
SelectAccess.Rdata_len = ADI_DEVICE_ACCESS_LENGTH0;

access_adxxxx.RegisterField = NULL;
access_adxxxx.ReservedValues = NULL;
access_adxxxx.ValidateRegister = NULL;
```

```
// Table to categorise ADxxxx
access_adxxxx.SelectAccess = & SelectAccess;
```

***pAdditionalinfo**

```
// To be used for future extensions (passed as NULL)
access_adxxxx.pAdditionalinfo = NULL;
```

Finally, the device driver should call the device access service function to access its internal registers. The device access service returns a code indicating the device access process was a success or failure.

```
// Call the device access service function
Result = adi_device_access (&access_adxxxx);
```

4.3. Device access command examples

4.3.1. Single register access

Consider device ADxxxx defined in section 4 & 4.1. To read/write a single register (in this case, DevReg2)

```
// Code example to access a single register
// define the structure to access a single device register
ADI_DEV_ACCESS_REGISTER Access_Reg;

// Load the device register address to be accessed
Access_Reg.Address = ADxxxx_DEVREG2;

//To read a single register
//-----------------------------------
//clear the Access_Reg.Data location
Access_Reg.Data = 0;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The register value will be read to location - Access_Reg.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_READ, (void *) &Access_Reg);
//-----------------------------------

//To configure a single register
//-----------------------------------------
//Load the configuration value to Access_Reg.Data location
Access_Reg.Data = 0x16;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The device register will be configured with the value in Access_Reg.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_WRITE, (void *) &Access_Reg);
//-----------------------------------------
```

4.3.2. Specific register field access

Consider device ADxxxx defined in section 4 & 4.1. To read/write a specific register field (in this case, Field 3 in DevReg1)

```
// Code example to access a specific register field
// define the structure to access a specific device register field
ADI_DEV_ACCESS_REGISTER_FIELD Access_Field;

// Load the device register address to be accessed
Access_Field.Address = ADxxxx_DEVREG1;
// Load the device register field location to be accessed
Access_Field.Address = ADxxxx_DEVREG1_FIELD1;

//To read a specific register field
//-----------------------------------------
//clear the Access_Field.Data location
Access_Field.Data = 0;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The register field value will be read to location - Access_Field.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_READ, (void *) &Access_Field);
//-----------------------------------------
```

```
//To configure a specific register field
//-------------------------------------------------
// Load the configuration value to Access_Field.Data location
Access_Field.Data = 2;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The device register field will be configured with the value in Access_Field.Data
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_WRITE, (void *) &Access_Field);
//-------------------------------------------------
```

4.3.3. Register table access

Consider device ADxxxx defined in section 4 & 4.1. To read/write table of device registers (in this case, three ADxxxx registers)

```
// Code example to access a table of selective device registers
// define the structure to access a table of device registers
ADI_DEV_ACCESS_REGISTER Access_Regs[ ] =
    { { ADxxxx_DEVREG1,  0 },       // register address to access, location to hold/holding its data
      { ADxxxx_DEVREG4,  0 },
      { ADxxxx_DEVREG6,  0 },
      { ADI_DEV_REGEND, 0 } };      // Device access delimiter (MUST be added to terminate the table)

//To read table of registers
//-------------------------------------
// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of registers listed in the table will be read to corresponding Data location in Access_Regs array
// i.e., value of ADxxxx_DEVREG1 will be read to Access_Regs[0].Data,
// value of ADxxxx_DEVREG4 to Access_Regs[1].Data and so on
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_READ, (void *)&Access_Regs[0]);
//-------------------------------------

//To configure a table of registers
//-----------------------------------------------
//Load corresponding register configuration values to Access_Regs Data locations
Access_Regs[0].Data = 0x16;
Access_Regs[1].Data = 0x1a;
Access_Regs[2].Data = 0x05;
// Application calls adi_dev_Control( ) function with corresponding command and value
//The registers listed in the table will be configured with corresponding Access_Regs[i].Data values
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_TABLE_WRITE, (void *) &Access_Regs[0]);
//-----------------------------------------------
```

4.3.4. Register Field table access

Consider device ADxxxx defined in section 4 & 4.1. To read/write table of device register fields (in this case, three ADxxxx register fields)

```
// Code example to access a table of selective device register fields
// define the structure to access a table of device register(s) fields
ADI_DEV_ACCESS_REGISTER_FIELD Access_Fields[ ] =
    { { ADxxxx_DEVREG2,  ADxxxx_DEVREG2_FIELD3, 0 },
      { ADxxxx_DEVREG6,  ADxxxx_DEVREG6_FIELD1, 0 },
      { ADxxxx_DEVREG2,  ADxxxx_DEVREG2_FIELD1, 0 },
      { ADI_DEV_REGEND, 0, 0 } }; // Device access delimiter (MUST be added to terminate the table)
```

```
//To read table of register(s) fields
//-------------------------------------------------
// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of register fields listed in the table will be read to
// corresponding Data location in Access_Fields array
// i.e., value of ADxxxx_DEVREG2_FIELD3 will be read to Access_Fields[0].Data,
// value of ADxxxx_DEVREG6_FIELD1 to Access_Fields[1].Data and so on
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_TABLE_READ, (void *)&Access_Fields[0]);
//-------------------------------------------------
```

```
//To configure table of register(s) fields
//-----------------------------------------------------------
//Load corresponding register field configuration values to Access_Fields Data locations
Access_Fields[0].Data = 3;
Access_Fields[1].Data = 6;
Access_Fields[2].Data = 1;
// Application calls adi_dev_Control( ) function with corresponding command and value
// Register fields listed in the table will be configured with corresponding Access_Fields[i].Data values
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_FIELD_TABLE_WRITE,
                                        (void *)&Access_Fields[0]);
//-----------------------------------------------------------
```

4.3.5. Register block access

Consider device ADxxxx defined in section 4 & 4.1. To read/write block of device registers (in this case, four ADxxxx registers starting from ADxxxx_DEVREG1)

```
// Code example to access a block of consecutive device registers
// define the structure to access a block of device registers
ADI_DEV_ACCESS_REGISTER_BLOCK Access_Block;

// load the number of registers to be accessed
Access_Block.Count = 4;
// load the starting address of the register block
Access_Block.Address = ADxxxx_DEVREG1;
// define a 'Count' sized array to hold register data read/written from/to the device
u16 Block_Data[4] = { 0 };
// load the start address of the above array to Access_Block data pointer
Access_Block.pData = &Block_Data[0];
```

```
//To read a block of registers
//----------------------------------------
// Application calls adi_dev_Control( ) function with corresponding command and value
// Present value of the registers in the given block will be read to corresponding Block_Data[ ] array
// value of ADxxxx_DEVREG1 will be read to Block_Data [0], ADxxxx_DEVREG2 to Block_Data[1],
// ADxxxx_DEVREG3 to Block_Data[2] and ADxxxx_DEVREG4 to Block_Data[3]
adi_dev_Control(DriverHandle, ADI_DEV_CMD_REGISTER_BLOCK_READ, (void *) &Access_Block);
//----------------------------------------------
```

//To configure a block of registers

```
//-----------------------------------------------
//Load corresponding register configuration values to Block_Data array
Block_Data [0] = 0x12;              // DevReg1 data
Block_Data [1] = 0x0A;              // DevReg2 data
Block_Data [2] = 0x21;              // DevReg4 data (DevReg3 is read-only)
Block_Data [3] = 0x0C;              // DevReg5 data

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