ANALOG DEVICES

USB NET2272 DEVICE DRIVER

DATE: 7 MAY 2010

Table of Contents

List of Tables

Document Revision History

Date	Description of Changes
01/06/2007	Initial Revision SG
07-May-2010	Additions to API detailed.

Table 1 – Revision History

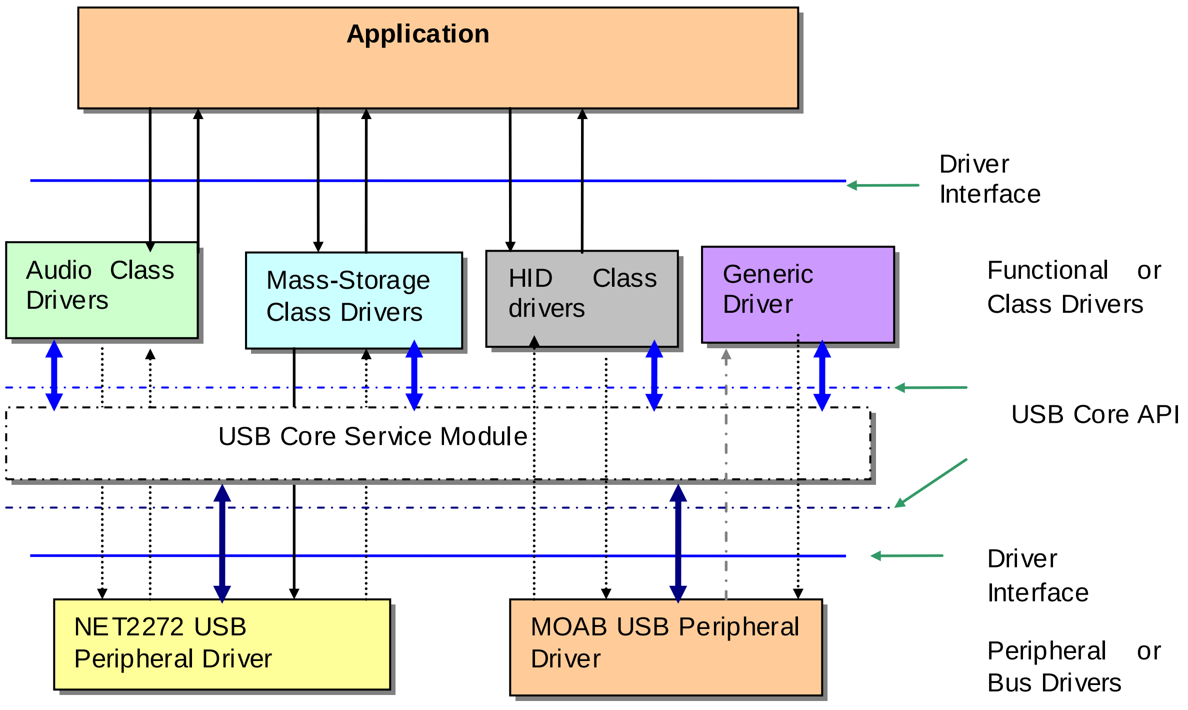
1. Overview

The overall USB driver stack architecture is depicted in the below figure. In this model the lowest level drivers are USB peripheral drivers, they interact with the USB hardware. Peripheral drivers offer primitive services like read, write and I/O controls to the upper level drivers and the USB core layer. All USB Peripheral drivers conform to the system services driver model. All interactions to the peripheral drivers are through the standard system services API. This abstraction and conformance to standard API allows confining the hardware specific details to the peripheral driver itself. Any USB driver conforming to the USB driver model can be easily plugged in reusing the upper level drivers and core.

USB Core implements USB protocol and provides services to the class drivers and the peripheral drivers. Critical functions of USB core include the device enumeration and constructing and managing various class configurations. Interactions with the USB core are via a standard USB Core API.

Class drivers are responsible for construction and managing the class specific configurations and data buffers. Class drivers conform to the system services driver model. USB applications typically interact with the class drivers.

Net2272 USB driver is a USB physical driver that interacts with the Net2272 peripheral and provides means to send and receive data over USB endpoints.


Application
Driver Interface
Audio Class Drivers
Mass-Storage Class Drivers
HID Class drivers
Generic Driver
Functional or Class Drivers
USB Core API
USB Core Service Module
Driver Interface
NET2272 USB Peripheral Driver
MOAB USB Peripheral Driver
Peripheral or Bus Drivers

Net2272 Device driver only operates in device mode.

Interaction between peripheral Driver and the USB Core Layer

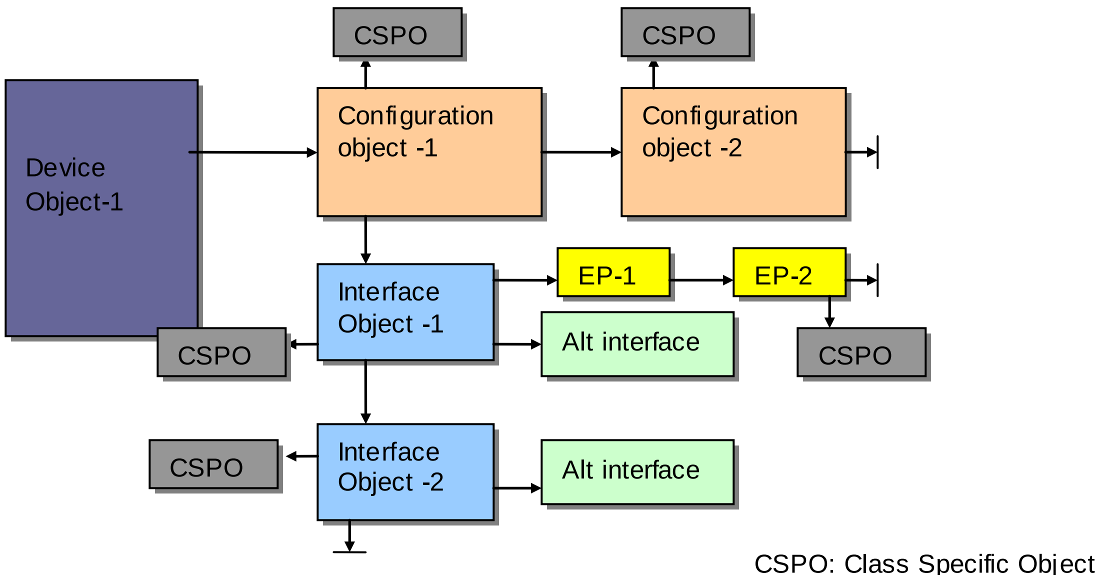
Various attributes of an USB device is specified through descriptors. Each USB device is represented with one device descriptor and one or more configuration, interface, endpoint descriptors. In USB peripheral mode class drivers construct the descriptor lists by using the services of the USB core layer. USB core layer maintains the devices configuration, interface and endpoint descriptor lists and supplies to the underlying peripheral driver.

USB endpoints provide communication means between two USB systems. In peripheral mode endpoints are created by the USB class drivers. Typically all USB applications interact with the class drivers. Class driver use the services of the USB core layer and also interact with the peripheral driver.

Organization of the various USB objects and descriptors:

Each of the below objects is a C- structure which also includes the associated descriptor structures.

Each object is also associated with class specific object which is used to represent the class specific structure. As class drivers encapsulate the class specific information, USB core or the underlying peripheral driver does not try to interpret the data in the class specific object. There is only one device object per device but more than one configuration objects can exists with class specific configuration. Each configuration can have multiple interfaces with possible alternate settings. Each interface can have interface specific configuration. Each interface can have multiple endpoints.



CSPO: Class Specific Object

2. Files

The files listed below comprise the device driver API and source files.

2.1. Include Files

The driver sources include the following include files:

- <services/services.h>
  - This file contains all definitions, function prototypes etc. for all the System Services.
- <drivers/adi_dev.h>
  - This file contains all definitions, function prototypes etc. for the Device Manager and general device driver information.
- <install-dir>\Blackfin\include\drivers\usb\controller\peripheral\plx\net2272\adi_usb_net2272.h
  - This file contains all the definitions, function prototypes for the Net2272 USB device driver.

2.2. Source Files

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

- <install-dir>\Blackfin\lib\src\drivers\usb\controller\peripheral\plx\net2272\adi_usb_net2272.c
  - This file contains the implementation of Net2272 peripheral driver.

3. Lower Level Drivers

Net2272 driver is the lowest level driver and there are no other lower level drivers.

4. Resources Required

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

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

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

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

In the following tables the System Services enumeration values are used to indicate the resources used. The values shown are the default values used. For custom arrangements, these can be overridden using command-value pairs passed to adi_dev_Conrtrol(). See Section 5.4.3 for details on the relevant commands.

4.1. Interrupts

The Net2272 IRQ signal is connected to a GPIO pin and managed by programmable flags in the Net2272 driver. Please refer to section 4.5 below for the default programmable flag assignments and associated reset interrupt priorities.

4.2. DMA

By default Net2272 driver uses memory DMA to transfer data to and from the Net2272 controller. Below memory dma channels were used by default. Users may change the default MDMA channel using ADI_USB_CMD_SET_DMA_CHANNEL i/o control command.

Processor	Memory DMA Channel	MDMA IVG level
ADSP-BF533	ADI_DMA_MDMA_0	IVG12
ADSP-BF537	ADI_DMA_MDMA_0	IVG12
ADSP-BF561	ADI_DMA_MDMA2_0	IVG12*
ADSP-BF518	ADI_DMA_MDMA_0	IVG12

* Please note, to enable the DMA interrupt to be handled it is necessary to override this value in the application with a priority higher than the Net2272 IVG level before enabling USB.

4.3. Timers

Not used.

4.4. Real-Time Clock

Not used.

4.5. Programmable Flags

By default Net2272 driver uses the following GPIO pin assignments and programmable flag Interrupts for the Net2272 interrupt signal. These defaults are aligned to the ADI extender cards listed in Section 7. Please refer to

Processor	**Net2272 IRQ**	**Programmable Flag Interrupt**	**Reset IVG Level**
ADSP-BF533	ADI_FLAG_PF10	ADI_INT_PFA	IVG13
ADSP-BF537	ADI_FLAG_PF7	ADI_INT_PORTFG_A	IVG13
ADSP-BF561	ADI_FLAG_PF15	ADI_INT_PF0_15_A	IVG11
ADSP-BF518	ADI_FLAG_PH3	ADI_INT_PORTH_INTA	IVG11

Other signals required use the following programmable flags by default. These are not interrupts.

Processor	**Net2272 Reset**	**Net2272 Clear**
ADSP-BF533	ADI_FLAG_PF11	Not used
ADSP-BF537	ADI_FLAG_PF6	Not used
ADSP-BF561	ADI_FLAG_PF11	ADI_FLAG_PF12
ADSP-BF518	ADI_FLAG_PF	Not used

The above assignments can be overridden with command

4.6. Pins

These follow the Programmable flag assignments shown in section 4.5.

5. Supported Features of the Device Driver

5.1. Directionality

The driver supports the dataflow directions listed in the table below. USB Endpoints are uni-directional and the direction is always specified in terms of the host. For example a windows or OTG host receives data on IN endpoint and sends data via OUT endpoint. In case of a USB peripheral an IN endpoint is used to send data and OUT endpoint is used to receive data. Depending on the host or peripheral mode read or write operations has to be performed. USB driver will assert if any illegal combination is used.

ADI_DEV_DIRECTION	Description
ADI_DEV_ DIRECTION_BIDIRECTIONAL	Supports both the reception of data and transmission of data through the device.

Table 2 – Supported Dataflow Directions

5.2. Dataflow Methods

The driver supports the dataflow methods listed in the table below.

ADI_DEV_MODE	Description
ADI_DEV_MODE_CHAINED	Supports the chained buffer method

Table 3 – Supported Dataflow Methods

5.3. Buffer Types

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

- ADI_DEV_1D_BUFFER
  - Linear one-dimensional buffer
  - pAdditionalInfo – ignored
  - **Reserved[4]** field of the buffer should have the destination endpoint number. Applications can obtain the endpoint information via the ADI_USB_CMD_ENUMERATE_ENDPOINTS control command to the associated class driver.

5.4. Command IDs

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

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

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

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

5.4.1. Device Manager Commands

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

- ADI_DEV_CMD_TABLE
  - Table of command pairs being passed to the driver
  - Value – ADI_DEV_CMD_VALUE_PAIR *
- ADI_DEV_CMD_END
  - Signifies the end of a command pair table
  - Value – ignored
- ADI_DEV_CMD_PAIR
  - Single command pair being passed
  - Value – ADI_DEV_CMD_PAIR *

5.4.2. Common Commands

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

- ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT
  - Determines if the device driver is supported by peripheral DMA
  - Value – u32 * (location where TRUE or FALSE is stored)

5.4.3. Device Driver Specific Commands

The command IDs listed below are supported and processed by the device driver. These command IDs are unique to this device driver. They represent all commands that are relevant to a USB Controller driver compatible with the ADI Blackfin USB Stack; some commands are thus required to simply return ADI_DEV_RESULT_NOT_SUPPORTED.

- ADI_USB_CMD_GET_DEVICE_ID
  - Returns the device identifier of the USB device. Each physical USB device has unique ID.
  - Value – u32* (location where the device id is stored)
- ADI_USB_CMD_ENABLE_USB
  - Enables USB device, which enables all USB interrupts. USB endpoints become ready to transmit or receive data.
  - Value – void
- ADI_USB_CMD_DISABLE_USB
  - Disables USB operation upon success.
  - Value – void
- ADI_USB_CMD_SET_STALL_EP
  - Stalls the passed endpoint.
  - Value – ENDPOINT_OBJECT* (pointer to the endpoint object)
- ADI_USB_CMD_CLEAR_STALL_EP
  - Clears a pending stall for the given endpoint.
  - Value – ENDPOINT_OBJECT* (pointer to the endpoint object)
- ADI_USB_CMD_SET_PF
  - Overrides the default programmable flag assignments for the Reset, IRQ, Clear and Set signals. This enables the driver to be used with a custom hardware arrangement.
  - Value – u32 * (The address of a 4 element u32 array containing the assignments in the order given above).
- ADI_USB_CMD_GET_PF

  - Returns the current programmable flag assignments in the passed array.
  - Value – u32 * (The address of a 4 element u32 array to store the values on return). See ADI_USB_CMD_SET_PF for details of array element assignments
- ADI_USB_CMD_SET_IVG
  - Overrides the default IVG level for the Net2272 IRQ signal.
  - Value – the IVG level which should be between 7 and 14, (13 if deferred callbacks are used at IVG14 or the driver is used in a VDK-based application).
- ADI_USB_CMD_GET_IVG
  - Returns the current IVG level for the Net2272 IRQ signal.
  - Value – u32* (location where the current IVG level is to be stored on return)
- ADI_USB_CMD_SET_DMA_CHANNEL
  - Overrides the default memory DMA channel.
  - Value – ADI_DMA_STREAM_ID
- ADI_USB_CMD_GET_DMA_CHANNEL
  - Returns the current memory DMA stream ID.
  - Value - ADI_DMA_STREAM_ID* (location where
- ADI_USB_CMD_SET_DMA_IVG
  - Overrides the default IVG level for the DMA interrupt.
  - Value – u32 (IVG level)
- ADI_USB_CMD_GET_DMA_IVG
  - Returns the current IVG level for the memory DMA interrupt.
  - Value – u32* (location where the current DMA IVG level is to be stored on return).
- ADI_USB_CMD_GET_BUFFER_PREFIX
  - Returns number of bytes allocated as prefix.
  - Value – u32* (location where the prefix value gets returned)
- ADI_USB_CMD_SET_BUFFER_PREFIX
  - Sets the buffer prefix.
  - Value – u32* (location where the buffer prefix is stored)
- ADI_USB_CMD_ENABLE_CNTRL_STATUS_HANDSHAKE
  - Initiates the USB hand-shake. Used only by the USB core. Applications should not use this I/O control.
  - Value – Not used.
- ADI_USB_CMD_SET_DEV_ADDRESS
  - Sets the device address. Used by the USB core. Applications should not invoke this I/O control
  - Value – u32 value that is the new device address
- ADI_USB_CMD_ACTIVATE_EP_LIST
  - Performs logical to physical endpoint binding. Applications should not invoke this I/O control. USB core uses it.
  - Value – ENDPOINT_OBJECT* (pointer to the list of active endpoint objects)
- ADI_USB_CMD_SET_DEV_MODE
  - Not supported, Returns ADI_DEV_RESULT_NOT_SUPPORTED. Only Supports MODE_DEVICE.
  - Value – n/a
- ADI_USB_CMD_GET_DEV_MODE
  - Returns the current device mode.
  - Value – location where device mode value is stored. MODE_DEVICE is the only supported mode.
- ADI_USB_CMD_BUFFERS_IN_CACHE
  - If data cache is enabled applications or class drivers has to inform the peripheral driver using this I/O control. With this data buffers will flushed and invalidated when required.
  - Value – Not used
- ADI_USB_CMD_GET_DEV_SPEED
  - Returns the operating speed of the device
    - **Valid values are as below:**
    - ADI_USB_DEVICE_SPEED_UNKNOWN
    - ADI_USB_DEVICE_SPEED_HIGH
    - ADI_USB_DEVICE_SPEED_FULL
    - ADI_USB_DEVICE_SPEED_LOW
  - Value – void pointer where current operating speed value is returned.

- ADI_USB_CMD_SET_DEV_SPEED
    - Not supported, Returns ADI_DEV_RESULT_NOT_SUPPORTED result code.

5.5. Callback Events

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

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

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

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

5.5.1. Common Events

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

5.5.2. Device Driver Specific Events

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

Each data endpoint is associated with a callback. This schema allows multiple class drivers to have class specific callback.
Events for the Data endpoint callbacks:

- ADI_USB_EVENT_DATA_RX
    - Notifies callback function about a completed read operation by the device driver.
    - Value – For chained or sequential I/O dataflow methods, this value is the CallbackParameter value that was supplied in the buffer that was passed to the adi_dev_Read() function.
- ADI_USB_EVENT_DATA_TX
    - Notifies callback function about a completed transmit operation
    - Value – The address of the buffer provided in the adi_dev_Write() function.
- ADI_USB_EVENT_PKT_RCVD_NO_BUFFER
    - Notifies the callback function that there is no receive buffer available to fill the incoming packet. Callback function may add a buffer using adi_dev_Read() call, else the packet will be dropped.
    - Value – Null.

Control endpoint (EP0) callback events are used by the USB core layer. All active configurations will be notified about certain events.

- ADI_USB_EVENT_SETUP_PKT
    - Notifies callback function about a received setup input packet. Gets triggered only in device mode.
    - Value – buffer that is associated with endpoint zero via adi_dev_Read.
    - Applications and class drivers will not be getting this event.
- ADI_USB_EVENT_START_OF_FRAME
    - Notifies callback function about a received start of frame.

    - Value – None.
    - All active configurations will be notified of this event. Disabled by default.
- ADI_USB_EVENT_ROOT_PORT_RESET
    - Notifies callback function about port reset.
    - Value – None.
    - All active configurations will be notified of this event
- ADI_USB_EVENT_RESUME
    - Notifies callback function about resume event.
    - Value – None.
    - All active configurations will be notified of this event
- ADI_USB_ EVENT_SUSPEND
    - Notifies callback function about suspend event. If USB is unplugged after enumeration is completed suspend event gets triggered.
    - Value – None.
    - All active configurations will be notified of this event

5.6. Return Codes

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

The return codes are divided into two sections. The first section describes return codes that are common to many device drivers. The next section describes driver specific return codes. The client should prepare to process each of the return codes described in these sections.

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

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

5.6.1. Common Return Codes

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

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

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

Return codes specific to TWI/SPI Device access service

- ADI_DEV_RESULT_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

5.6.2. Device Driver Specific Return Codes

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

6. Opening and Configuring the Device Driver

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

6.1. Entry Point

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

- ADI_USB_NET2272_Entrypoint

6.2. Default Settings

The table below describes the default configuration settings for the device driver. If the default values are inappropriate for the given system, the application should use the command IDs listed in the table to configure the device driver appropriately. Any configuration settings not listed in the table below are undefined.

Item	Default Value	Possible Values	Command ID
Buffers in cache	OFF	True or False	ADI_USB_CMD_BUFFERS_IN_CACHE
Device Mode	MODE_DEVICE	MODE_DEVICE	ADI_USB_CMD_SET_DEV_MODE

Table 4 – Default Settings

6.3. Additional Required Configuration Settings

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

ADI_USB_CMD_ENABLE_USB activates the USB devices and enables USB interrupts. So any configuration settings and default buffer allocations to the endpoints have to be done prior to issuing the enable command.

Item	Possible Values	Command ID
Dataflow method	true	ADI_USB_CMD_ENABLE_USB

Table 5 – Additional Required Settings

7. Hardware Considerations

The following extender cards are available for the ADI evaluation boards as follows:

Evaluation Board	**Extender Card Part Number**
ADSP-BF533 EZ-KIT LITE	ADDS-USBLAN-EZEXT
ADSP-BF537 EZ-KIT LITE	ADDS-USBLAN-EZEXT
ADSP-BF561 EZ-KIT LITE	ADDS-USBLAN-EZEXT
ADSP-BF518F EZ-BRD	ADZS-BFSHUSB-EZEXT