ANALOG DEVICES

USB HDRC DEVICE DRIVER

DATE: 8 AUGUST 2008

Table of Contents

List of Tables

Document Revision History

Date	Description of Changes
01/06/2007	Initial Revision SG
8 August 2008	Updated to cover BF52x support and sections added to address implementation issues

Table 1 – Revision History

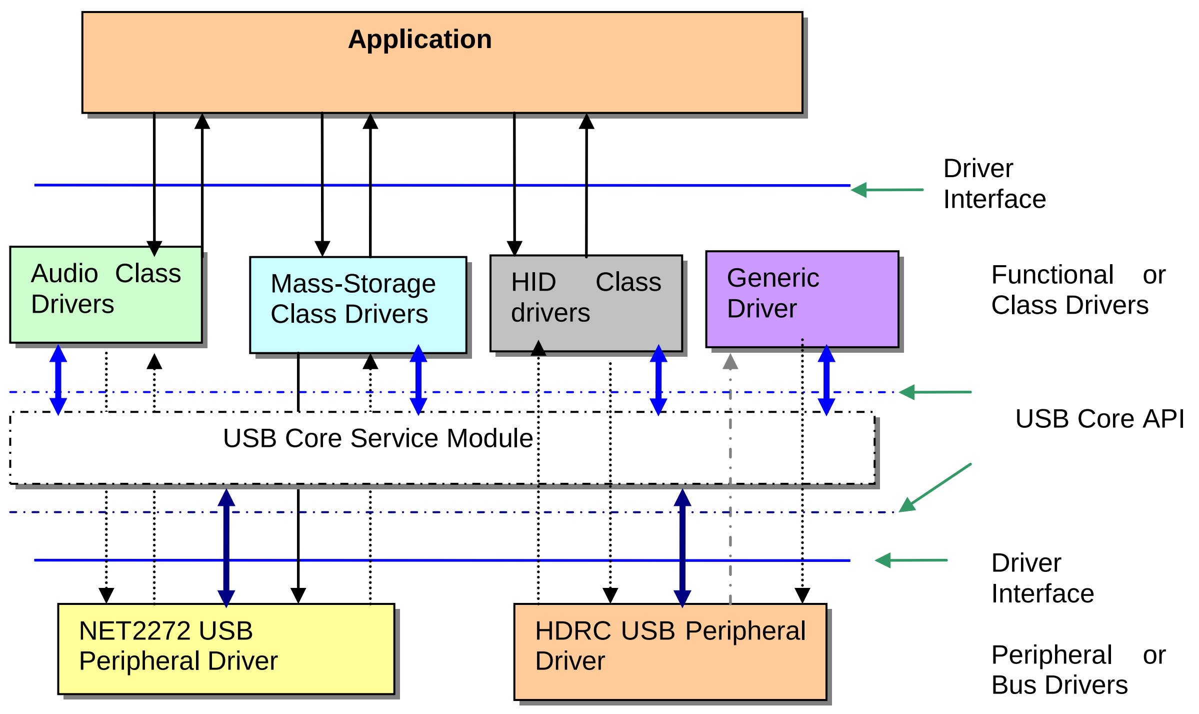
Overview

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.

The HDRC USB driver is a USB physical driver that interacts with the USB Controller on BF54x & BF52x processors and provides the 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 Service Module
USB Core API
Driver Interface
NET2272 USB Peripheral Driver
HDRC USB Peripheral Driver
Peripheral or Bus Drivers

HDRC Driver works on ADSP-HDRC EZ-kit. The driver includes both Device and OTG host functionality. By default the driver boots up as neither host nor device. Applications must use adi_usb_SetDeviceMode() core API to force the driver to boot in either OTG host mode or device mode. Any changes to that of default values (Device Mode, Device Speed etc) have to be done before enabling the USB.

Interaction between peripheral Driver and the USB Core Layer

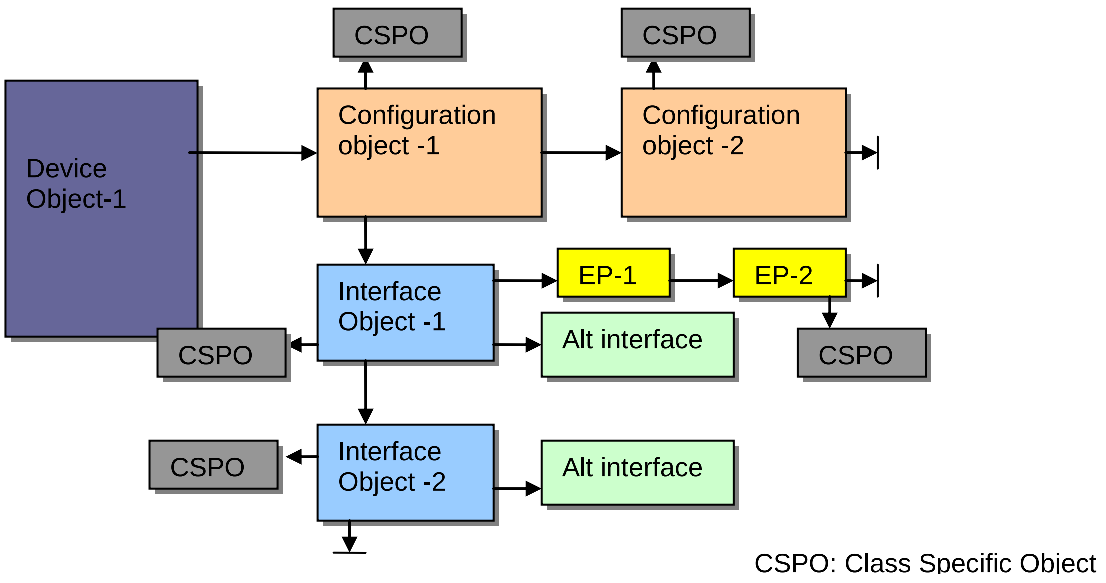
Various attributes of an USB device is specified through descriptors. Each USB device is represented with one device descriptor and one or more configuration, interface, endpoint descriptors. In USB peripheral mode class drivers construct the descriptor lists by using the services of the USB core layer.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



1. Files

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

1.1. 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\usb\controller\otg\adi\hdrc\adi_usb_hdrc.h

  - This file contains all the definitions, function prototypes for the HDRC USB device driver.

1.2. 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\otg\adi\hdrc\adi_usb_hdrc.c
  - This file contains the implementation of the HDRC driver (both Device and OTG host)

2. Lower Level Drivers

There are no other lower level drivers.

3. Resources Required

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

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

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

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

3.1. Interrupts

The HDRC USB device driver uses four USB interrupts: ADI_INT_USB_INT0, ADI_INT_USB_INT1, ADI_INT_USB_INT2 and ADI_INT_USB_DMAINT. The HDRC driver hooks all these interrupts at the ik_ivg11 priority level using one interrupt handler.

3.2. DMA

The HDRC USB controller has its own dedicated DMA channels, which are not managed by the DMA manager. The HDRC driver thus programmes the USB DMA registers directly. This gives the necessary flexibility for getting the maximum throughput using DMA mode zero or DMA mode one as necessary. The HDRC driver uses DMA to transfer data from the endpoint FIFOS to the memory. Although supported this driver does not use programmed I/O transfers for performance reasons.

3.3. Timers

Not used.

3.4. Real-Time Clock

Not used.

3.5. Programmable Flags

One flag is required in host mode to select the USB VBUS signal to power connected devices. In device mode this flag remains unasserted. For custom boards the flag pin can be overridden with the appropriate command-value pair. See section 4.4.3 for further details.

EZ-KIT	**Flag**	**Assertion polarity**
ADSP-BF548 EZ-KIT	ADI_FLAG_PE7	High
ADSP-BF527 EZ-KIT	ADI_FLAG_PG13	High
ADSP-BF526 EZ-KIT	ADI_FLAG_PG13	Low

4. Supported Features of the Device Driver

4.1. 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

4.2. 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

4.3. 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.

4.4. Command IDs

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

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

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

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

4.4.1. Device Manager Commands

Shortcut to MediaPlayerDemoBF527.lnk 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 *

4.4.2. Common Commands

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

- 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)

4.4.3. Device Driver Specific Commands

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

- ADI_USB_CMD_GET_DEVICE_ID
  - Returns the USB device identifier.
- ADI_USB_CMD_ENABLE_USB
  - Enables USB, interrupts, USB endpoints become active to transmit or receive data.

- ADI_USB_CMD_DISABLE_USB
  - Disables USB operation upon success.
- ADI_USB_CMD_SET_STALL_EP
  - Stalls the passed endpoint.
- ADI_USB_CMD_CLEAR_STALL_EP
  - Clears a pending stall for the given endpoint.
- ADI_USB_CMD_SET_IVG
  - Configures the Interrupt levels of the USB interrupts.
- ADI_USB_CMD_GET_IVG
  - Returns the current Interrupt levels of the USB interrupts.
- ADI_USB_CMD_GET_BUFFER_PREFIX
  - Returns number of bytes allocated as prefix.
- ADI_USB_CMD_SET_BUFFER_PREFIX
  - Sets the buffer prefix.
- ADI_USB_CMD_ENABLE_CNTRL_STATUS_HANDSHAKE
  - Initiates the USB hand-shake. Used only by the USB core. Application should not invoke this I/O control.
- ADI_USB_CMD_SET_DEV_ADDRESS
  - Sets the device address. Used by the USB core. Applications should not invoke this I/O control
- ADI_USB_CMD_ACTIVATE_EP_LIST
  - Performs logical to physical endpoint binding. Applications should not invoke this I/O control. USB core uses it.
- ADI_USB_CMD_OTG_REQ_IN_TOKEN
  - Requests IN token for receiving data in case of OTG host mode. Applications should not invoke this I/O control. USB core uses it.
- ADI_USB_CMD_SEND_ZERO_LEN_PKT
  - Sends zero length packet in case of OTG host mode. Applications should not invoke this I/O control. USB core uses it.
- ADI_USB_CMD_SET_DEV_MODE
  - Sets the device mode, MODE_DEVICE or MODE_OTG_HOST. Mode has to be selected when USB is disabled.
- ADI_USB_CMD_GET_DEV_MODE
  - Returns the current device 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.
- ADI_USB_CMD_GET_DEV_SPEED
  - Returns the operating speed of the device.
- ADI_USB_CMD_SET_DEV_SPEED
  - Sets the operating speed of the device. Typically used to force high speed device to operate at full speed.
- 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. Not currently in use; will return failure result code.
  - 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_IVG
    - Configures the Interrupt levels of the USB interrupts.
    - Value – ADI_USB_BF54x_IVGS* (pointer to the USB IVG level structure)
- ADI_USB_CMD_GET_IVG
    - Returns the current Interrupt levels of the USB interrupts.
    - Value – ADI_USB_BF54x_IVGS* (pointer to the USB IVG level structure)
- ADI_USB_CMD_SET_PF
    - Assigns the GPIO flag to be used to select VBUS to be driven
    - Value - ADI_FLAG_ID value for required pin
- ADI_USB_CMD_GET_PF
    - Returns the VBUS selection GPIO pin
    - Value – location to hold the ADI_FLAG_ID value.
- 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_OTG_REQ_IN_TOKEN
    - Requests IN token for receiving data in case of OTG host mode. Applications should not invoke this I/O control. USB core uses it.
    - Value – Not used
- ADI_USB_CMD_SEND_ZERO_LEN_PKT
    - Sends zero length packet in case of OTG host mode. Applications should not invoke this I/O control. USB core uses it.
    - Value – Not used
- ADI_USB_CMD_SET_DEV_MODE
    - Sets the device mode, MODE_DEVICE or MODE_OTG_HOST. Mode has to be selected when USB is disabled.
    - Value – MODE_OTG_HOST or MODE_DEVICE
- ADI_USB_CMD_GET_DEV_MODE
    - Returns the current device mode.
    - Value – location where device mode value is stored. The value can be either MODE_OTG_HOST or MODE_DEVICE
- ADI_USB_CMD_BUFFERS_IN_CACHE
    - If data cache is enabled for an application, this command can be used to instruct the USB HDRC driver to use software instructions to synchronize the cached memory prior to DMA transfer. It is the responsibility of the class drivers to issue this command.
    - Value – Not used
- ADI_USB_CMD_ENTER_TEST_MODE
    - Instructs the controller to enter the given test mode
    - Value – the required test mode.
- ADI_USB_CMD_PEEK_EP_FIFO
    - Requests the value of the RX count register for the given endpoint number
    - Value – on entry contains the required endpoint number; on return it contains the appropriate RX Count.

- ADI_USB_CMD_STOP_EP_TOKENS
    - Stops the generation of tokens for the given endpoint number.
    - Value – contains the required endpoint number.
- 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
    - Sets the operating speed of the USB device
    - Value – should be one of the following.
        - ADI_USB_DEVICE_SPEED_HIGH
        - ADI_USB_DEVICE_SPEED_FULL
        - ADI_USB_DEVICE_SPEED_LOW

4.5. Callback Events

All callback events are posted to the appropriate callback function for an endpoint as configured by the USB core and OTG modules. The device driver callback function is not used; thus the ClientCallback argument in the call to adi_dev_Open() is ignored.

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

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

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

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

4.5.1. Common Events

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.

4.5.2. Device Driver Specific Events

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

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. But applications will also be notified for 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.
    - Value – None.
    - All active configurations will be notified of this event
- ADI_USB_EVENT_VBUS_TRUE
    - Notifies callback function about VBus above threshold.
    - Value – None.
    - All active configurations will be notified of this event
- ADI_USB_EVENT_VBUS_FALSE
    - Notifies callback function about VBus below threshold.
    - Value – None.
    - All active configurations will be notified of this event
- ADI_USB_EVENT_RX_STALL
    - Notifies callback function that an IN transfer has stalled. Host mode only.
    - Value – Endpoint Number
- ADI_USB_EVENT_RX_ERROR
    - Notifies callback function of an error in receiving data from the devic. Host mode only.
    - Value – Endpoint Number
- ADI_USB_EVENT_RX_NAK_TIMEOUT
    - Notifies callback function that the device has repeatedly issued NAKs up to the point where the controller times out. Host mode only.
    - Value – Endpoint Number

4.6. Return Codes

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

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

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

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

4.6.1. Common Return Codes

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

- ADI_DEV_RESULT_SUCCESS
  - The function executed successfully.
- ADI_DEV_RESULT_NOT_SUPPORTED
  - The function is not supported by the driver.
- ADI_DEV_RESULT_DEVICE_IN_USE
  - The requested device is already in use.
- ADI_DEV_RESULT_NO_MEMORY
  - There is insufficient memory available.
- ADI_DEV_RESULT_BAD_DEVICE_NUMBER
  - The device number is invalid.
- ADI_DEV_RESULT_DIRECTION_NOT_SUPPORTED
  - The device cannot be opened in the direction specified.
- ADI_DEV_RESULT_BAD_DEVICE_HANDLE
  - The handle to the device driver is invalid.
- ADI_DEV_RESULT_BAD_MANAGER_HANDLE
  - The handle to the Device Manager is invalid.
- ADI_DEV_RESULT_BAD_PDD_HANDLE
  - The handle to the physical driver is invalid.
- ADI_DEV_RESULT_INVALID_SEQUENCE
  - The action requested is not within a valid sequence.
- ADI_DEV_RESULT_ATTEMPTED_READ_ON_OUTBOUND_DEVICE
  - The client attempted to provide an inbound buffer for a device opened for outbound traffic only.
- ADI_DEV_RESULT_ATTEMPTED_WRITE_ON_INBOUND_DEVICE
  - The client attempted to provide an outbound buffer for a device opened for inbound traffic only.
- ADI_DEV_RESULT_DATAFLOW_UNDEFINED
  - The dataflow method has not yet been declared.
- ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE
  - The dataflow method is incompatible with the action requested.
- ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE
  - The device does not support the buffer type provided.
- ADI_DEV_RESULT_CANT_HOOK_INTERRUPT
  - The Interrupt Manager failed to hook an interrupt handler.
- ADI_DEV_RESULT_CANT_UNHOOK_INTERRUPT
  - The Interrupt Manager failed to unhook an interrupt handler.
- ADI_DEV_RESULT_NON_TERMINATED_LIST

  - The chain of buffers provided is not NULL terminated.
- ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED
  - No callback function was supplied when it was required.
- ADI_DEV_RESULT_REQUIRES_UNIDIRECTIONAL_DEVICE
  - Requires the device be opened for either inbound or outbound traffic only.
- ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE
  - Requires the device be opened for bidirectional traffic only.

Return codes specific to TWI/SPI Device access service

- ADI_DEV_RESULT_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

4.6.2. 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.

5. Opening and Configuring the Device Driver

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

5.1. Entry Point

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

- ADI_USBDRC_Entrypoint

5.2. Default Settings

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

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

Table 4 – Default Settings

5.3. Additional Required Configuration Settings

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

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

6. Hardware Considerations

6.1. GPIO pin

Of particular importance is the requirement for a GPIO pin to be used to activate an external FET switch. Depending on the manufacturer of the FET switch, the polarity of the signal required to activate the switch may differ. For example, an ADI FET switch has been chosen for the BF526 EZ-KIT which requires a different polarity (low) to that used on the BF548 and BF527 EZ-KITs (high). For use with a custom board, it is not yet possible to configure the polarity of the signal used within the driver.

Please note that the use of a high polarity for switching VBUS will result in the EZ-KIT providing 5V off board when the Blackfin is in the hibernate state. The design of the BF526 EZ-KIT addresses this concern, which is particularly relevant as the board can be powered from a battery.

The driver requires the following EZ-KIT settings for host mode.

BF548 EZ-KIT	BF527 EZ-KIT	BF526 EZ-KIT
SW16.4 ON	SW13.2 OFF SW13.6 ON	SW20.2 ON SW20.6 OFF

In addition for the BF548 EZ-KIT it has been found that SW16.4 should be set to OFF to gain stability in peripheral mode.

6.2. USB mini A/B connector

To comply with the USB OTG specification, boards making use of the Blackfin USB OTG controller must provide a mini socket, capable of receiving both mini type A and B plugs.

When using the USB HDRC driver in host mode it is importand to use a mini type A plug. If access to USB flash drives is required, it will be necessary to use a lead with a mini type A plug on one end and a type A socket on the other.

When using the USB HDRC driver in preipheral mode a mini type B plug should be used. This plug comes as standard on most leads that connect mobile phones, PDAs & some cameras to PCs.

The Analog Devices EZ-KITs come with the necessary connections for both host and peripheral modes of operation.

7. Implementation Issues

7.1. Cache Coherency

Please note that the BF54x & BF52x processors – as indeed with all Blackfin processors to date – do not provide hardware support to maintain coherency of cached data when this data is updated via DMA. In other words, the cache functions of the processor are unaware that data has either been changed behind the scenes by the DMA, or is required to be synchronized with the actual memory before transmitting the data via DMA.

The USB HDRC driver can be configured to use software coherency methods so that the DMA buffers supplied to the driver can be placed in cacheable memory. However, it is the responsibility of a class/host driver to turn this function on, via the ADI_USB_CMD_BUFFERS_IN_CACHE command. Please refer to Section 4.4.3 for further details. If this feature is not used, or not supplied in a class driver then the application can either use this command or must ensure that all DMA buffers are placed in non-cacheable memory.

7.2. Workarounds for Silicon Anomalies

7.2.1. L1 DMA Buffers on early BF523/5/7 processors

Data corruption and lock up occur on silicon rev 0.1 and below of the BF523/5/7 processors and silicon rev 0.0 of the BF522/4/6 processors when USB DMA buffers are located in L3 memory, cached or otherwise. It is important to locate these buffers in L1 memory, and it is the responsibility of the application to ensure that this is the case. (Please note, that the File System Service Physical Interface Driver for USB provides an 8K L1 intermediate buffer for this purpose).

7.3. Untested features

The USB HDRC driver has been tested for Hi-Speed Control and Bulk transfers only. The following features have not been tested:

1. Full speed transfers. These transfers are typically required when connecting the Blackfin to a USB 1.1 controller or device. Such controllers may be present in older PCs. Whilst the support is provided in the USB HDRC driver, none of the ADI supplied class drivers support such transfers and hence it has not been tested.

2. Similarly, low-speed transfers are untested.

3. Isochronous transfers. Whilst the support is provided in the USB HDRC driver, ADI has no class drivers that support such transfers and hence it has not been tested.

Analog Devices Inc. cannot respond to support queries regarding these features.

7.4. Unsupported features

7.4.1. OTG support

The USB HDRC driver can be used in either host or peripheral mode. It is currently not possible to dynamically switch from peripheral to host and visa versa. Neither does it support the Host Negotiation Protocol. In that sense, full OTG support is not currently provided.