ADI USB HOST MASS STORAGE PHYSICAL INTERFACE DRIVER

DATE: 13 NOVEMBER 2007

Table of Contents

Table of Figures

Document Revision History

Date	Description of Changes
19 September 2007	Initial Draft
13 November 2007	Revised.

Table 1 - Revision History

1 Overview

This document describes the functionality of ADI USB Physical Interface Driver (PID) that conforms to the specification required for integration within the Analog Devices' File System Service (FSS). The USB PID is appropriate for all USB flash drives that utilize Logical Block Address (LBA) Sector numbers to define locations on the physical media and are USB logo certified for High Speed USB.

2 Quick Start Guide

2.1 Reference Chart for System Services Initialization:

The following table details the System Services & Device Driver resources as required to be set in the adi_ssl_init.h header file.

DMA channels	None – The USB OTG has its own DMA controller
Device Drivers	Three: one for PID, one for class driver and one for controller driver.
Semaphores	Two.
Flag Callbacks	None.
Interrupts[1]	
Peripheral ID	**Default IVG**
`ADI_INT_USB_INT0`	11
`ADI_INT_USB_INT1`	11
`ADI_INT_USB_INT2`	11
`ADI_INT_USB_DMAINT`	11

2.2 Registering the USB PID with the File System Service

To add the USB PID to the FSS, include the USB PID header file, adi_usb.h in the application source code, optionally choosing to accept the default definition structure, `ADI_USB_Def`, e.g:

```
#define _ADI_USB_DEFAULT_DEF_
#include <drivers/pid/usb/adi_usb.h>
```

Then add the following command-value pair to the FSS configuration table before calling adi_fss_Init():

```
{ ADI_FSS_CMD_ADD_DRIVER, (void*)&ADI_USB_Def },
```

For details on providing a custom definition please refer to Section 9.1.

[1] Depending on other active interrupts in the application you will need to allocate at least one secondary interrupt handler for the following default levels.

2.3 Dynamic Memory Requirements

The following table details the amount of dynamic memory required for an associated operation.

Operation	Size (bytes)
Device Instance. (One instance per ATA/USB chain)	64
Master boot record – retained for the duration of volume detection only	512

3 Files

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

3.1 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.

- **`<services/fss/adi_fss.h>`**
  This file contains all definitions, function prototypes etc. for the File System Service.

- **`<drivers/pid/usb/adi_usb.h>`**
  This file contains all definitions, function prototypes etc. for the appropriate Physical Interface Driver.

- **`<string.h>`**
  This file all definitions, function prototypes etc. for the memory copy functions.

3.2 Source Files

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

- **`Blackfin\lib\src\drivers\pid\usb\adi_usb.c`**

All source code is written in ‘C’. There are no assembly level functions in this driver.

4 Lower Level Drivers

The USB Host Mass Storage Class Driver:

- **Blackfin\lib\src\drivers\usb\class\otg\mass_storage\adi_usb_msd_class_host.c**
- **Blackfin\lib\src\drivers\usb\class\otg\mass_storage\adi_usb_msd_host_support.c**

BF54x OTG controller driver:

- **Blackfin\lib\src\drivers\usb\controller\otg\adi\hdrc\adi_usb_hdrc.c**

In addition the ADI USB protocol stack is used by the class and controller drivers. This stack is include in libusb548.dlb and libusb527.dlb.

5 Resources Required

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

Unless explicitly noted in the sections below, the USB PID 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.

All memory requirements other than data structures created on the stack are met dynamically via calls to the centralized memory management functions in the FSS, `_adi_fss_malloc()`, `_adi_fss_realloc()`, and `_adi_fss_free()`. These functions are wrappers for either the default libc functions, `heap_malloc()`, `heap_realloc()` and `heap_free()`, or for application specific functions as defined upon configuration of the File System Service. In this way the implementer can chose to supply memory management functions to organize a fixed and known amount of memory.

Two heap types are supported by the File System Service, a *cache* heap for data buffers such as the source or target of DMA transfers, and a *general* heap for house-keeping data such as instance data. Upon configuration of the USB PID implementers can only specify the heap index for the *cache* heap; the USB PID makes use of the general heap defined in the FSS for all housekeeping structures. If no cache heap is defined the USB PID will use the FSS general heap.

The value of the *cache* heap index is set using the command-value pair

```
{ ADI_FSS_CMD_SET_CACHE_HEAP_ID, (void*)CacheHeapIndex }
```

Where `CacheHeapIndex` is either the index in the `heap_table_t heap_table` array (see the <project>_heaptab.c file), or that obtained from the call to `heap_install`:

```
static u8 myheap[1024];
#define MY_HEAP_ID 1234
:
int CacheHeapIndex = heap_install((void *)&myheap, sizeof(myheap), MY_HEAP_ID );
```

The use of customizable heaps may be dependent on the development environment. If the chosen environment does not support customizable heaps then the FSS routines will have been modified to ignore the heap index argument.

The following table details the amount of dynamic memory required for an associated operation.

Operation	Size (bytes)
Device Instance. (One instance per ATA/USB chain)	64
Master boot record – retained for the duration of volume detection only	512

5.1 Interrupts

The following interrupts are used:

Peripheral ID	Default IVG	Description
`ADI_INT_USB_INT0`	11	Internal to USB controller.
`ADI_INT_USB_INT1`	11	Internal to USB controller
`ADI_INT_USB_INT2`	11	Internal to USB controller
`ADI_INT_USB_DMAINT`	11	Internal to USB controller

Please note that the current implementation of the controller driver precludes changing these priorities with calls to `adi_int_SICSetIVG()` prior to driver initialization.

5.2 DMA

The USB PID uses the OTG host controller DMA channels.

5.3 Timers

No timers are used by the USB PID.

5.4 Semaphores

The USB PID requires two semaphores, one for a Lock Semaphore to maintain exclusive access to the PID from one process at a time, and one for signaling completion of internal data transfers. The Semaphore Service must be used to create and manipulate all semaphores.

5.5 Real-Time Clock

Use of the RTC Service is not required by this class of driver.

5.6 Programmable Flags

No programmable flags are used by the USB PID.

5.7 Pins

The USB OTG controller driver is configured to use the default pins assigned to the on-chip USB OTG controller.

6 Supported Features of the Device Driver

This section describes what features are supported by the device driver.

6.1 Directionality

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

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

Table 2 - Supported Dataflow Directions

6.2 Dataflow Methods

A PID can only support the driver `ADI_DEV_MODE_CHAINED` dataflow method. When used within the FSS this is applied automatically. If the PID is operated in standalone mode it is essential to send the following command-value pair to the PID, before activating the driver:

```
{ ADI_DEV_CMD_SET_DATAFLOW_METHOD, (void*)ADI_DEV_MODE_CHAINED },
```

6.3 Buffer Types

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

- **ADI_DEV_1D_BUFFER**
  Linear one-dimensional buffer. This is enveloped by the FSS Super Buffer Structure (Section 7.5)
  - `CallbackParameter` – This will always contain the address of the FSS Super Buffer structure.
  - `ProcessedFlag` - This field is not used in the USB PID.
  - `pAdditionalInfo` – This field is not used in the USB PID.

6.4 Command IDs

This section enumerates the commands that are supported/required by the USB PID. The commands are divided into three sections. The first section describes commands that are supported directly by the Device Manager. The second section describes common Device Driver commands that the USB PID supports. The next section describes commands specific to PID drivers. Finally, the last subsection details commands specific to the USB PID.

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 an `ADI_DEV_DEVICE_HANDLE` type that uniquely identifies the device driver. This handle is provided to the client on return from 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.

6.4.1 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_VALUE_PAIR *`
- **`ADI_DEV_CMD_SET_SYNCHRONOUS`**
  - Enables/disables synchronous mode for the driver
  - Value – `TRUE`/`FALSE`
- **`ADI_DEV_CMD_GET_INBOUND_DMA_CHANNEL_ID`**
  - Returns the DMA channel ID value for the device driver's inbound DMA channel
  - Value – `u32 *` (location where the channel ID is stored)
- **`ADI_DEV_CMD_GET_OUTBOUND_DMA_CHANNEL_ID`**
  - Returns the DMA channel ID value for the device driver's outbound DMA channel.
  - Value – `u32 *` (location where the channel ID is stored)
- **`ADI_DEV_CMD_SET_INBOUND_DMA_CHANNEL_ID`**
  - Sets the DMA channel ID value for the device driver's inbound DMA channel
  - Value – `ADI_DMA_CHANNEL_ID` (DMA channel ID)
- **`ADI_DEV_CMD_SET_OUTBOUND_DMA_CHANNEL_ID`**
  - Sets the DMA channel ID value for the device driver's outbound DMA channel
  - Value – `ADI_DMA_CHANNEL_ID` (DMA channel ID)
- **`ADI_DEV_CMD_SET_DATAFLOW_METHOD`**
  - Specifies the dataflow method the device is to use. The list of dataflow types supported by the device driver is specified in section 6.2.
  - Value – `ADI_DEV_MODE` enumeration

6.4.2 Common Device Driver 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 the USB PID.

- **ADI_DEV_CMD_GET_INBOUND_DMA_PMAP_ID**
  - Returns the PMAP ID for the device driver's inbound DMA channel.
  - Value – `u32 *` (location where the PMAP value is stored)
- **ADI_DEV_CMD_GET_OUTBOUND_DMA_PMAP_ID**
  - Returns the PMAP ID for the device driver's outbound DMA channel
  - Value – `u32 *` (location where the PMAP value is stored)
- **ADI_DEV_CMD_SET_DATAFLOW**
  - Enables/disables dataflow through the device
  - Value – `TRUE`/`FALSE`
- **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)
- **ADI_DEV_CMD_FREQUENCY_CHANGE_PROLOG**
  - Notifies device driver immediately prior to a CCLK/SCLK frequency change. If the SCLK frequency increases then the device drivers timings need to be adjusted prior to frequency change, otherwise the driver must pause its activity somehow.
  - Value – `ADI_DEV_FREQUENCIES *` (new frequencies)
- **ADI_DEV_CMD_FREQUENCY_CHANGE_EPILOG**
  - Notifies device driver immediately following a CCLK/SCLK frequency change. If the SCLK frequency decreases then the device drivers timings need to be adjusted subsequent to frequency change, otherwise the driver must resume its activity somehow.
  - Value – `ADI_DEV_FREQUENCIES` * (new frequencies)

6.4.3 PID Specific Commands

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

- **ADI_FSS_CMD_GET_BACKGRND_XFER_SUPPORT**
  - Requests the USB PID to return `TRUE` or `FALSE` depending on whether the device supports the transfer of data in the background. The USB PID currently returns `FALSE`.
  - Value – Client provided location to store result.
- **ADI_FSS_CMD_GET_DATA_ELEMENT_WIDTH**
  - Requests the USB PID to return the width (in bytes) that defines each data element. The USB PID returns 2 since the BF54x ATA interface only supports 16 bit (2 bytes) DMA buffers.
  - Value – Client provided location to store result.

- **ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE**
  - Requests the USB PID to grant a Lock Semaphore to give the calling module exclusive access to the PID data transfer functions.
  - Value – `NULL`.
- **ADI_FSS_CMD_RELEASE_LOCK_SEMAPHORE**
  - Requests the PID to release the Lock Semaphore granted in response to the **ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE** command.
  - Value – `NULL`.
- **ADI_FSS_CMD_SET_CACHE_HEAP_ID**
  - Instructs the USB PID instance to use the given Heap Index for any dynamically allocated data buffers. The default heap Index for such buffers defaults to -1, indicating that the FSS General Heap is to be used.
  - Value – the Index of the required heap.
- **ADI_PID_CMD_GET_FIXED**
  - Requests the USB PID to return `TRUE` or `FALSE` depending on whether the device is to be regarded as Fixed or removable. The USB PID returns `TRUE`.
  - Value – Client provided location to store result.
- **ADI_PID_CMD_MEDIA_ACTIVATE**
  - Activates the ATA/USB device, configuring it for use. This may include assigning certain programmable flags and programming the PORT MUX registers as necessary.
  - Value – `TRUE` to activate, `FALSE` to deactivate.
- **ADI_PID_CMD_POLL_MEDIA_CHANGE**
  - Instructs the USB PID to check the status of the device for the removal or insertion of media. If the driver detects that media has been removed it issues the `ADI_FSS_EVENT_MEDIA_REMOVED` callback event to the Device Manager Callback function. If the driver detects that media has been inserted it issues the `ADI_FSS_EVENT_MEDIA_INSERTED` callback event, (Section 6.5.2).
  - Value – `NULL`.
- **ADI_PID_CMD_DETECT_VOLUMES**
  - Instructs the USB PID to discover the volumes/partitions available on the media. For each volume detected the PID issues the `ADI_FSS_EVENT_VOLUME_DETECTED` event, passing the pointer to the salient volume information as the third argument. (See Section 6.5.2).
  - Value – `NULL`.
- **ADI_PID_CMD_SEND_LBA_REQUEST**
  - Requests the USB PID to command the device to read/write a number of sectors from/to a given LBA start sector.
  - Value – Address of the `ADI_FSS_LBA_REQUEST` structure containing the above information.
- **ADI_PID_CMD_ENABLE_DATAFLOW**
  - Instructs the USB PID to take the necessary steps to begin/stop dataflow. Please note that the `{ADI_DEV_CMD_SET_DATAFLOW,(void*)TRUE}` command will not reach the PDD part of the USB PID after the first call unless a `{ADI_DEV_CMD_SET_DATAFLOW,(void*)FALSE}` command is received in-between.

- Value - TRUE/FALSE.

- **ADI_PID_CMD_SET_DIRECT_CALLBACK**
  - Provides the address of a callback function to call directly (i.e. non-deferred) upon media insertion/removal and volume detection events. (See Section 6.5.2).
  - Value – the address of the direct callback function.

6.5 Callback Events

This section enumerates the callback events the USB PID generates. The events are divided into two sections. The first section describes events that are common to many device drivers. The next section describes FSS specific event IDs. The FSS defines a callback function that supports the required Events. In standalone use, the implementer should prepare a callback function to process each event described in these two sections.

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

- `ClientHandle`. Except for callbacks to the direct callback function this `void*` parameter will be the `DeviceHandle` (3rd) argument passed to the `adi_pdd_Open` function of the PID. For direct callbacks it must be the address of this argument.
- `EventID`
  This is a `u32` data type that specifies the event ID. See below.
- `Value`
  This parameter is a `void*` whose value is context sensitive to the specific event ID.

Most callbacks are directed to the Device Manager provided callback function specified as the last argument, `DMCallback`, passed to the `adi_pdd_Open` function of the PID. The Device Manager will post a deferred callback if a valid DCB queue handle was passed to `adi_dev_Open()`. Support for deferred callbacks is governed upon configuration of the FSS.

The exceptions to this rule are the `ADI_FSS_EVENT_MEDIA_INSERTED`, `ADI_FSS_EVENT_MEDIA_REMOVED` and `ADI_FSS_EVENT_VOLUME_DETECTED` events, where it is required in the context of the File System Service that non-deferred callbacks must be used. The function to call directly is set with the `ADI_PID_CMD_SET_DIRECT_CALLBACK` command by the FSS. Please note that in this case the `ClientHandle` to pass to the direct callback function is the address of the `DeviceHandle` argument.

For standalone use, when the `ADI_PID_CMD_SET_DIRECT_CALLBACK` command is omitted, the USB PID will use the usual Device Manager Route.

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

6.5.1 Common Events

The events described in this section are common to many device drivers. The list below details the only common event ID currently supported by the USB PID.

- **ADI_DEV_EVENT_BUFFER_PROCESSED**

Notifies callback function that a chained I/O buffer has been processed by the device driver.
Value – This value is the `CallbackParameter` value that was supplied in the buffer that was passed to the `adi_dev_Read()` or `adi_dev_Write()` function.

6.5.2 FSS Specific Events

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

- **`ADI_FSS_EVENT_MEDIA_INSERTED`**
  This event is issued in response to the `ADI_PID_CMD_POLL_MEDIA_CHANGE` command upon detection that media has been inserted[2].
  Value – The address of a data location. On issue of the callback this location contains the Device Number of the device (zero if not applicable – see `ADI_PID_CMD_SET_NUMBER_DEVICES` command). On return from the callback the location contains a result code. If the result code returned is `ADI_FSS_RESULT_SUCCESS`, the USB PID will regard the media as being present and correctly accounted for by the FSS.
- **`ADI_FSS_EVENT_MEDIA_REMOVED`**
  This event is issued in response to the `ADI_PID_CMD_POLL_MEDIA_CHANGE` command upon detection that media has been removed. It is also issued automatically via the callback from the class driver upon the asynchronous detection of media removal.
  Value – The address of a data location. On issue of the callback this location contains the Device Number of the device. On return from the callback, the contents have no significance.
- **`ADI_FSS_EVENT_VOLUME_DETECTED`**
  This event is issued in response to the `ADI_PID_CMD_DETECT_VOLUMES` command upon detection of a valid volume/partition.
  Value – The address of an `ADI_FSS_VOLUME_DEF` structure defining the volume:
  - `FileSystemType` - The File system type, as defined in the adi_fss.h header file under the title “Enumerator for known File System types”. See the FSS Implementation document for further details.
  - `StartAddress` - The Sector (LBA value) of the first sector in the volume.
  - `VolumeSize` - The size of the volume in sectors.
  - `SectorSize` - The size in bytes of each sector on the volume.
  - `DeviceNumber` - The number of the device in a chain of devices. This should be zero if not applicable.

  This structure must be regarded as volatile by the FSS (or application callback in standalone mode), and as such can be declared on the stack within the USB PID. Its values need to be copied in the FSS or application callback prior to returning control to the USB PID if they are to be retained.

- **`ADI_PID_EVENT_DEVICE_INTERRUPT`**
  This event is issued by the USB PID once all data pertaining to an LBA request is processed.

[2] Please not that it is not currently possible due to hardware limitations to asynchronously detect when media is inserted.

Value – The address of the Buffer structure associated with the event. This must be the value located in the `pBuffer` field of the associated LBA request structure.

6.6 Return Codes

All API functions of the USB PID return a status code 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` or `ADI_FSS_RESULT_SUCCESS` indicates success, while any other value indicates an error or some other informative result. The values `ADI_DEV_RESULT_SUCCESS` and `ADI_FSS_RESULT_SUCCESS` are 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
}
```

6.6.1 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 the USB PID.

- **`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_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_BIDIRECTIONAL_DEVICE**
  Requires the device be opened for bidirectional traffic only.

6.6.2 FSS Specific Return codes used by the USB PID driver

The following return codes are defined in the <services/fss/adi_fss.h> header file:

- **ADI_FSS_RESULT_NO_MEDIA**
  No media is detected, or the Identify command fails.
- **ADI_FSS_RESULT_NO_MEMORY**
  There was insufficient memory to complete a request. Usually as a result of a call to _adi_fss_malloc().
- **ADI_FSS_RESULT_MEDIA_CHANGED**
  The media has changed.
- **ADI_FSS_RESULT_FAILED**
  General failure.
- **ADI_FSS_RESULT_NOT_SUPPORTED**
  The requested operation is not supported by the PID.
- **ADI_FSS_RESULT_SUCCESS**
  General Success.

7 Data structures

7.1 Device Driver Entry Points, ADI_DEV_PDD_ENTRY_POINT

This structure is used in common with all drivers that conform to the ADI Device Driver model, to define the entry points for the device driver. It is defined in the USB PID source module, adi_usb.c, and declared as an extern variable in the USB PID header file, adi_usb.h, where its presence is guarded from inclusion in the PID source module as follows:

- In the source module and ahead of the `#include` statement for the header file, define the macro, `__ADI_USB_C__`.
- In the header file, guard the extern declaration:

```
#if !defined(__ADI_USB_C__)
extern ADI_DEV_PDD_ENTRY_POINT ADI_USB_EntryPoint;
:
#endif
```

7.2 Command-Value Pairs, ADI_DEV_CMD_VALUE_PAIR

This structure is used in common with all drivers that conform to the ADI Device Driver model, and is used primarily for the initial configuration of the driver. The USB PID must support all three methods of passing command-value pairs:

- `adi_dev_control( …, ADI_DEV_CMD_TABLE, (void*)<table-address> );`
- `adi_dev_control( …, ADI_DEV_CMD_PAIR, (void*)<command-value-pair-address> );`
- `adi_dev_control( …, <command>, (void*)<associated-value );`

No default table is declared in the USB PID header file, adi_usb.h.

7.3 Device Definition Structure, ADI_FSS_DEVICE_DEF

This structure is used to instruct the FSS how to open and configure the USB PID. It's contents are essentially the bulk of the items to be passed as arguments to a call to `adi_dev_Open()`. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
    u32                     DeviceNumber;
    ADI_DEV_PDD_ENTRY_POINT *pEntryPoint;
    ADI_DEV_CMD_VALUE_PAIR  *pConfigTable;
    void                    *pCriticalRegionData;
    ADI_DEV_DIRECTION       Direction;
    ADI_DEV_DEVICE_HANDLE   DeviceHandle;
    ADI_FSS_VOLUME_IDENT    DefaultMountPoint;
} ADI_FSS_DEVICE_DEF;
```

Where the fields are assigned as shown in the following table:

`DeviceNumber`	This defines which peripheral device to use. This is the `DeviceNumber` argument required for a call to `adi_dev_Open()`. This value is ignored by the USB PID.
`pEntryPoint`	This is a pointer to the device driver entry points and is passed as the `pEntryPoint` argument required for a call to `adi_dev_Open()`. For the USB PID its value should be assigned to `&ADI_USB_EntryPoint`.
`pConfigTable`	This is a pointer to the table of command-value pairs to configure the USB PID; the default value for the USB PID is `NULL`.
`pCriticalRegionData`	This is a pointer to the argument that should be passed to the System Services `adi_int_EnterCriticalRegion()` function. This is currently not used and should be set to `NULL`.
`Direction`	This is the `Direction` argument required for a call to `adi_dev_Open()`. For the USB PID this value should be `ADI_DEV_DIRECTION_BIDIRECTIONAL`.
`DeviceHandle`	This is the location - used internally - to store the Device Driver Handle set on return from a call to `adi_dev_Open()`. It should be set to `NULL` prior to initialization.
`DefaultMountPoint`	This is the default drive letter to be used for volumes managed by the USB PID. Setting this field will ensure that the same drive letter is used each time a USB device is inserted. The default definition in the adi_usb.h header file leaves this field blank.

A default instantiation of this structure is declared in the USB PID header file, adi_usb.h, and guarded against inclusion in the PID Source module, and will only be available in an application module if the developer defines the macro, `_ADI_USB_DEFAULT_DEF_`:

```
#if !defined(__ADI_USB_C__)
:
#if defined(_ADI_USB_DEFAULT_DEF_)
static ADI_FSS_DEVICE_DEF ADI_USB_Def = { … };
:
#endif
:
#endif
```

7.4 Volume Definition Structure, ADI_FSS_VOLUME_DEF

This structure is used within the USB PID to communicate to the FSS the presence of a usable volume or partition. An address to a global instantiation of the structure is returned as the third callback argument sent to the FSS along with the `ADI_FSS_EVENT_VOLUME_DETECTED` event. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
    u32 FileSystemType;
    u32 StartAddress;
    u32 VolumeSize;
```

```
    u32 SectorSize;
    u32 DeviceNumber;
} ADI_FSS_VOLUME_DEF;
```

Where the fields are assigned as shown in the following table:

FileSystemType	The unique identifier for the type of file system. Valid types are declared in an anonymous enum in the FSS header file. For Most USB devices this will be ADI_FSS_FSD_TYPE_FAT.
StartAddress	The starting sector of the volume/partition in LBA format.
VolumeSize	The number of sectors contained in volume/partition.
SectorSize	The number of bytes per sector used by the USB PID.
DeviceNumber	This is used to indicate the device number on a chain of devices. This value is set to 0 by the USB PID.

The FSS will regard this structure as volatile and will make a copy of its contents.

7.5 The FSS Super Buffer Structure, ADI_FSS_SUPER_BUFFER

A *Super Buffer* is used to envelope the ADI_DEV_1D_BUFFER structure. Since this, ADI_FSS_SUPER_BUFFER, structure has the ADI_DEV_1D_BUFFER structure as its first member, the two structures share addresses, such that

- The address of the Super buffer can be used in calls to adi_dev_Read/Write, and
- Where understood the *super* buffer can be de-referenced and its contents made use of.

At each stage of the submission process, from File Cache to FSD to PID, the super buffer gains pertinent information along the way. The fields are defined in the following table and are color coded such that red are the fields that the File Cache sets, green are the fields that an FSD sets, and blue are the fields that a PID sets. The LBA Request is set by the FSD for requests originating from both the cache and the FSD, or in the PID for its own internal requests.

Please note that for use outside the context of the file system service, all calls to adi_dev_Read() or adi_dev_Write() with the USB PID device handle must use the address of a valid ADI_FSS_SUPER_BUFFER structure.

The originator of the Super buffer will zero the fields that are not appropriate.

The definition of the structure is:

```
typedef struct ADI_FSS_SUPER_BUFFER{
    ADI_DEV_1D_BUFFER       Buffer;
    struct adi_cache_block  *pBlock;
    u8                      LastInProcessFlag;
    ADI_FSS_LBA_REQUEST     LBARequest;
    ADI_SEM_HANDLE          SemaphoreHandle;
    ADI_FSS_FILE_DESCRIPTOR *pFileDesc;
    ADI_DCB_CALLBACK_FN     FSDCallbackFunction;
    void                    *FSDCallbackHandle;
    ADI_DCB_CALLBACK_FN     PIDCallbackFunction;
```

```
    void                    *PIDCallbackHandle;
} ADI_FSS_SUPER_BUFFER;
```

Where the fields are defined as:

Buffer	The ADI_DEV_1D_BUFFER structure required for the transfer. Please note that this is not a pointer field. This is only set by the USB PID if it is originating the data transfer request.
SemaphoreHandle	The Handle of the Semaphore to be posted upon completion of data transfer. This is only set by the USB PID if it is originating the data transfer request, when it is set to the value stored in the USB PID instance data. See section below for use of semaphores.
LBARequest	The ADI_FSS_LBA_REQUEST structure for the associated buffer. See section 7.6 for details.
pBlock	Used in the File Cache. Its value remains unchanged by the USB PID. For internal USB PID transfers it is set to NULL.
LastinProcessFlag	Used in the File Cache. Its value remains unchanged by the USB PID. For internal USB PID transfers it is set to NULL.
pFileDesc	Used in the File Cache. Its value must remain unchanged by the USB PID. For internal USB PID transfers it is set to NULL.
FSDCallbackFunction	This handle is reserved for use with FSDs. For internal USB PID transfers it is set to NULL.
FSDCallbackHandle	This handle is reserved for use with FSDs. For internal USB PID transfers it is set to NULL.
PIDCallbackFunction	The USB PID assigns the address of the callback function to be invoked upon transfer completion events.
PIDCallbackHandle	The USB PID assigns the address of a pertinent structure to be passed as the first argument in the call to the function defined by the PIDCallbackFunction field.

7.6 LBA Request, ADI_FSS_LBA_REQUEST

This structure is used to pass a request for a number of sectors to be read from the device. The address of the LBARequest field in the associated ADI_FSS_SUPER_BUFFER structure (section 7.5) should be used. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct ADI_FSS_LBA_REQUEST {
    u32                     SectorCount;
    u32                     StartSector;
    u32                     DeviceNumber;
    u32                     ReadFlag;
    ADI_FSS_SUPER_BUFFER *pBuffer;
} ADI_FSS_LBA_REQUEST;
```

Where the fields are assigned as shown in the following table:

SectorCount	The number of sectors to transfer.
StartSector	The Starting sector of the block to transfer in LBA format.
DeviceNumber	The Device Number on the chain. This must be 0 for the USB PID.
ReadFlag	A Flag to indicate whether the transfer is a read operation. If so, then its value will be 1. If a write operation is required its value will be 0.
pBuffer	The address of the associated ADI_FSS_SUPER_BUFFER sub-buffer.

8 Data Transfer

In describing the data transfer procedure it is important to make the distinction between *device* events (initiated by the physical mass storage device) and *host* events (initiated by the software). As far as the USB PID is concerned, data transfer is active from the receipt of an LBA request to transfer a number of sectors and the completion of the transfer. This is termed a *DRQ block* after its ATA origins. On the other hand, the *host* considers the data transfer completion event as the point when it receives a callback upon completion of each ADI_DEV_1D_BUFFER.

Upon a *host* transfer completion event, the USB PID will automatically issue the ADI_DEV_EVENT_BUFFER_PROCESSED event via the Device Manager part of the device driver. Upon completion of each *DRQ block* the USB PID will issue the ADI_USB_PID_EVENT_DEVICE_INTERRUPT event. These callbacks are made via the Device Manager with the following arguments:

1 The DeviceHandle argument, supplied as the third argument passed to adi_pdd_Open().

2 The appropriate event code.

3 The address of pBuffer value in the LBA request structure.

In reply to these events, the FSS will make a call into the USB PID using the PIDCallbackFunction and PIDCallbackHandle fields of the ADI_FSS_SUPER_BUFFER structure:

```
(pSuperBuffer->PIDCallbackFunction)(
                                    pSuperBuffer->PIDCallbackHandle,
                                    Event,
                                    pArg );
```

In this function the USB PID will do what is required in each of the two events. Furthermore, in response to the ADI_USB_PID_EVENT_DEVICE_INTERRUPT event, the USB PID will release the USB PID Lock Semaphore and post the USB PID Semaphore *only* if the SemaphoreHandle value of the ADI_FSS_SUPER_BUFFER equals that of the USB PID Semaphore handle.

Where a chain of such buffers defines contiguous data on the media, and a single LBA request has been sent to the USB PID, to cover the chain or parts thereof, there will be several *host* transfer completion events to the one *device* transfer completion event (DRQ block). The USB PID locks access to the driver for the duration of the DRQ block, by maintaining a Lock Semaphore.

The process of issuing the request (usually by a File System Driver) is as follows:

1. Acquire Lock Semaphore from the USB PID passing the command-value pair,

```
{ ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE, NULL },
```

2. The LBA request for a first buffer in the chain is submitted to the USB PID by passing the command-value pair, e.g.:

```
{ ADI_USB_PID_CMD_SEND_LBA_REQUEST, (void*)&pSuperBuffer->LBARequest> },
```

   The USB PID assigns the `PIDCallbackFunction` and `PIDCallbackHandle` (Section 7.5) fields of the `ADI_FSS_SUPER_BUFFER` structure (Section 7.5) pointed to by the `pBuffer` field of the LBA Request structure and stores a copy of the LBA request structure in its instance data.

3. Then the FSD submits the required buffer chain to the USB PID via a call to `adi_dev_Read()` or `adi_dev_Write()`, e.g.

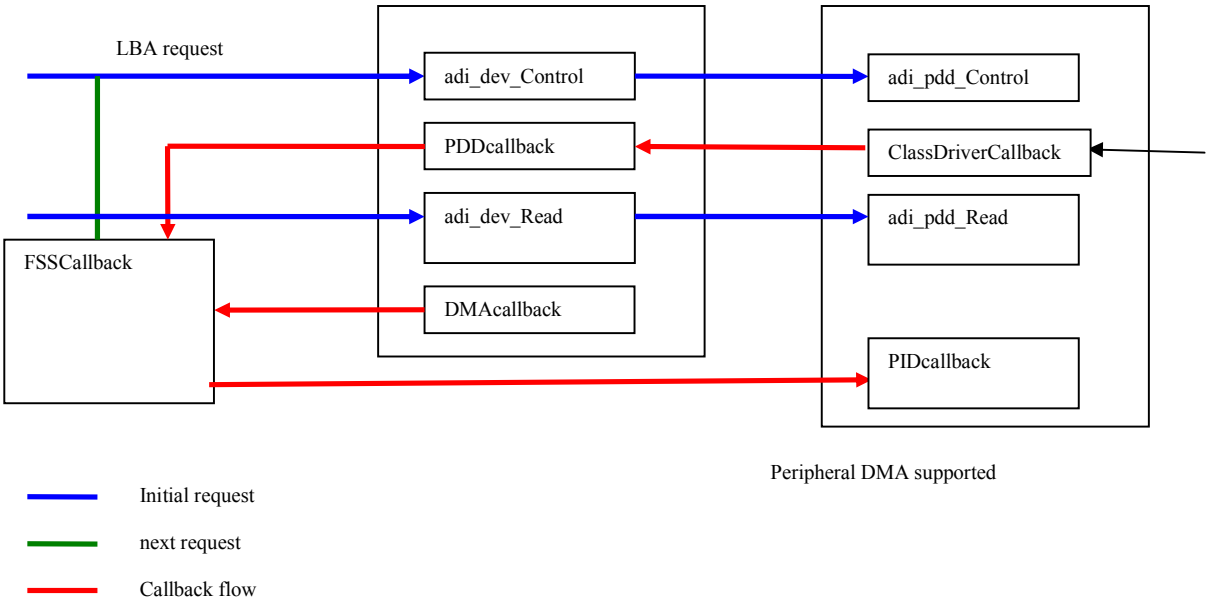
```
adi_dev_Read{…, ADI_DEV_1D, (ADI_DEV_BUFFER*)pSuperBuffer },
```

4. Data flow is enabled by sending the following command to the USB PID

```
{ ADI_USB PID_CMD_ENABLE_DATAFLOW, (void*)TRUE},
```

The Lock Semaphore acquired in stage 1 is released by the FSD either upon completion of the DRQ block for a single buffer (no chain) or upon completion of the DRQ block of the last sub-buffer in the chain.

The diagram below illustrates the command and callback flow of the USB PID. Please note that the *next request*s are handled by the appropriate FSD, but detail is omitted here for clarity.



9 Opening and Configuring the USB Physical Interface Driver

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

9.1 Registering the USB PID with the File System Service

To add the USB PID to the FSS, an instance of the `ADI_FSS_DEVICE_DEF` structure, e.g. `ADI_USB_Def`, must be defined (Section 7.3) and its address passed to the adi_fss_init() function as part of the FSS configuration table:

```
{ ADI_FSS_CMD_ADD_DRIVER, (void*)&ADI_USB_Def },
```

This structure will require the address of the USB PID entry point structure, `ADI_USB_EntryPoint` (Section 7.1), which is defined in the USB PID header file, <drivers/pid/usb/adi_usb.h>. An example configuration table (Section 7.2) and definition structure could be:

```
ADI_DEV_CMD_VALUE_PAIR ADI_USB_ConfigTable [] = {
            { ADI_FSS_CMD_SET_CACHE_HEAP_ID,   (void*)1 },
            { ADI_DEV_CMD_END,                 NULL     },
};

ADI_FSS_DEVICE_DEF ADI_USB_Def = {
            0,
            &ADI_USB_EntryPoint,
            ADI_USB_ConfigTable,
            NULL,
            ADI_DEV_DIRECTION_BIDIRECTIONAL,
            'C'
};
```

Alternatively, the default definition structure, defined in the USB PID header file, can be used by defining the `_ADI_USB_DEFAULT_DEF_` macro ahead of including the header file:

```
#define _ADI_USB_DEFAULT_DEF_
#include <drivers/pid/usb/adi_usb.h>
```

In the above definition, the default configuration table is not required so its address is set to `NULL` in the `ADI_FSS_DEVICE_DEF` structure.

Please note that the FSS will endeavor to apply the specified default mount point drive letter to this device and retain it through media changes. If a default drive letter is not required this value can be set to `NULL`. If the requested letter is not available at any stage then the FSS will assign the next available drive letter, starting from “c”.

9.2 Procedure for Opening the USB PID

The File System Service (FSS) will automatically open the USB PID by issuing a call to `adi_dev_Open()` in response to it being registered with the FSS (See Section 9.1). The arguments to this call are supplied by the `ADI_FSS_DEVICE_DEF` structure (section 7.3).

If successful, the remaining commands are received in the following order:

1. ADI_DEV_CMD_SET_DATAFLOW_METHOD – The dataflow method is set to ADI_DEV_MODE_CHAINED as mandatory for all FSS Device Drivers.
2. ADI_DEV_CMD_TABLE – here the address of the configuration table defined by the user and assigned to the pConfigTable field of the ADI_FSS_DEVICE_DEF structure (section 7.3) is passed to the USB PID for configuration.
3. ADI_PID_CMD_MEDIA_ACTIVATE – The USB PID initializes the lower level device drivers and requests device enumeration.
4. ADI_PID_CMD_SET_DIRECT_CALLBACK – The address of a callback function is specified to be called upon detection of media insertion/removal and also upon detection of a usable partition or volume.
5. ADI_PID_CMD_POLL_MEDIA_CHANGE – The USB PID is instructed to detect the presence of appropriate media.

   If media is detected the USB PID issues a live callback event to the direct callback function as defined above. This callback uses the following arguments:

   - The address of the DeviceHandle argument, supplied as the third argument passed to adi_pdd_Open().
   - The ADI_FSS_EVENT_MEDIA_INSERTED event.
   - The address of a u32 variable. The contents of this variable are set by the USB PID to the Device Number of the device on the chain for which media is detected, if appropriate. On return from the callback this variable will be set to an appropriate result code, either ADI_FSS_RESULT_FAILED or ADI_FSS_RESULT_SUCCESS, the latter value indicating that the FSS has correctly handled the detected media.
6. ADI_PID_CMD_DETECT_VOLUMES - In response to media insertion the FSS will instruct the PID to detect usable volumes/partitions on the device number identified by that associated argument.

   Upon detection of a valid volume, the PID issues a live callback to the direct callback function. This callback uses the following arguments:

   - The address of the DeviceHandle argument, supplied as the third argument passed to adi_pdd_Open().
   - The ADI_FSS_EVENT_VOLUME_DETECTED event.
   - The address of an ADI_FSS_VOLUME_DEF structure defining the volume. Please refer to Section 7.4 for details about the definition and assignment of this structure.

9.3 Initialization when used standalone

In cases where the USB PID is to be used outside the context of the FSS, for instance when partitioning of media is required, or where the embedded application is a USB peripheral application (where the File System support is provided by the Host PC), it may be necessary to initialize the USB PID separately from the context of the File System Service. This section details what is required.

The device driver definition structure, ADI_USB_Def, (Section 7.3) provides most of the requirements for the call to adi_dev_Open() to open the USB PID device driver:

```
Result = adi_dev_Open(
                        <DeviceManagerHandle>,
                        ADI_USB_Def.pEntryPoint,
                        ADI_USB_Def.DeviceNumber,
                        &ADI_USB_Def.DeviceHandle,
                        &ADI_USB_Def.DeviceHandle,
                        ADI_USB_Def.Direction,
                        <DMAManagerHandle>,
                        <DCBQueueHandle>,
                        <Callback-function>
                                );
```

The other arguments need to be supplied: The <DeviceManagerHandle> and <DMAManagerHandle> are those obtained from the usual initialization of the System Services & Device Manager. The <DCBQueueHandle> is the handle of the DCB queue if callbacks to <Callback-function> from the USB PID are to be deferred, (recommended).

Next, the USB PID needs to be configured with the required configuration settings (Section 9.5) and any optional settings required, for instance to over-ride the defaults settings (Section 9.4).

The <Callback-function> passed in the call to adi_dev_Open() will be called, not from the PDD section of the USB PID, but from the device manager part of the device driver. If the callback queue handle, <DCBQueueHandle>, has been assigned then this call will be deferred. However, the procedure for media detection requires a live-callback to either the same callback function (as is the case when initialized within the FSS framework) or to a separate function. Whichever it is, it must be registered with the PID with the ADI_PID_CMD_SET_DIRECT_CALLBACK command as described in Section 9.2.

The callback function(s) must handle the following events. In all these events the first argument in the callback is the address of a location containing the PID Device Handle, which will be the same value as given in the call to adi_dev_Open(), namely &ADI_USB_Def.DeviceHandle; the Event will be one of the following and the third argument is interpreted as required, and detailed below.

1. **ADI_FSS_EVENT_MEDIA_INSERTED**. The third argument is the address of a location containing the device number (0) of the device for which media is detected. On return it must contain a result code, indicating whether the callback has been handled successfully.
2. **ADI_FSS_EVENT_MEDIA_REMOVED**. The third argument has no meaning in this event. The action to take will depend on the purpose of the application.
3. **ADI_FSS_EVENT_VOLUME_DETECTED**. The third argument is the address of an ADI_FSS_VOLUME_DEF structure defining the volume. Please refer to Section 7.4 for details about the definition and assignment of this structure. The action to take will depend on the purpose of the application.
4. **ADI_DEV_EVENT_BUFFER_PROCESSED**. This is the host transfer completion event. Please refer to Section 8 for further details.

Further action may be required dependent on the application. For instance if the `pNext` field of the buffer is non-zero, and the `SectorCount` value of the LBA request of the next sub-buffer is non-zero then action may be required to queue the next LBA request with the PID, as is the case when used within the FSS framework. Please refer to Section 8 for further details.

5 **`ADI_PID_EVENT_DEVICE_INTERRUPT`**. This is the device transfer completion event. This is treated identically to the `ADI_DEV_EVENT_BUFFER_PROCESSED` event, as detailed in the previous point.

9.4 Default Settings

The table below describes the default configuration settings for the USB PID.

Item	Default Value	Possible Values	Command ID
Cache Heap Index	`-1`	The heap index to use for the allocation of data tansfer buffers.	`ADI_FSS_CMD_SET_CACHE_HEAP_ID`

Table 3 - Default Settings

9.5 Additional Required Configuration Settings

In addition to the possible overrides of the default driver settings, the USB PID responds to the following commands issued from the FSS as detailed below. The following table does not itemize the mandatory commands used by the FSS to communicate to the USB PID Driver (see Section 6.4.3 for further details).

Item	Possible Values	Command ID
Dataflow method	See section 6.2	`ADI_DEV_CMD_SET_DATAFLOW_METHOD`

Table 4 – Additional Required Settings