GENERIC PHYSICAL INTERFACE DRIVER DESIGN (PID) DOCUMENT FOR INCLUSION IN THE FILE SYSTEM SERVICE (FSS)

DATE: 20 JULY 2007

Table of Contents

Table of Figures

Document Revision History

Date	Description of Changes
6 June 2006	Initial Draft
16 June 2006	Added removable media events.
11 July 2006	Updated to current design spec.
23 August 2006	Updated to revised design spec and added more details for opening and configuring Media Device PIDs.
1 November 2006	Changed control command for partition information and updated configuration details.
6 February 2007	Updated interface to reflect improved functionality within the File System Framework.
15 February 2007	Added prototypes for the FSS memory management functions, which have changed to include heap index parameter. Added command to assign the heap index for dynamically allocated data caches.
30 April 2007	Changed format of document and added new sections.
12 June 2007	Updated to reflect redesign of the PID.
20 July 2007	Replaced ‘mutex’ with ‘lock semaphore’ throughout.

Table 1 - Revision History

1 Overview

This document describes the functionality of a Generic Physical Interface Driver (PID) that conforms to the specification required for integration within the Analog Devices' File System Service (FSS). To be termed a PID, a device driver for accessing mass storage media must adhere to the requirements of this document.

In addition to aiding the design of compliant drivers, this document also serves as a template for specific PID documentation.

In the following sections, where a PID identifier string is required, either in a file or data variable identifier, the following convention is used:

- {ns} – represents the namespace part of the ident, eg. adi.
- {iface} – represents the interface part of the ident, eg. atapi
- {pid-ident} ={ns}_{iface} – represents a lower case string, e.g.: adi_atapi
- If any of the above are in uppercase, e.g. {PID-IDENT} the string is the same as above but in upper case string, e.g.: ADI_ATAPI

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.

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

- **<drivers/pid/{iface}/{pid-ident}.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.

2.2 Source Files

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

- **<Blackfin/lib/src/drivers/ata/{iface}/{pid-ident}.c>**

This file contains all the source code for the File System Device Driver. All source code is written in ‘C’. There are no assembly level functions in this driver.

3 Lower Level Drivers

<Dependent on the target Blackfin processor family and connectivity, an on-chip peripheral driver may be utilized. For example, a serial optical drive as found in consumer products would require both the SPORT device driver for data transfer and the TWI device driver for control. Replace this section with details of the lower level drivers required.>

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.

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. To use these functions in a PID, the following statements must be included in the PID source file:

```
extern void *_adi_fss_malloc( int id, size_t size );
extern void _adi_fss_free( int id, void *p );
extern void *_adi_fss_realloc( int id, void *p, size_t size );
```

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 a PID, implementers can only specify the heap index for the *cache* heap; a PID must make use of the general heap defined in the FSS. Thus for all *general* heap usage, the PID should pass **`-1`** as the `id` argument, which the FSS will interpret as a request to use the *general* heap index stored by the FSS. Likewise, the default value for the PID's *cache* heap index should be `-1`.

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.

4.1 Interrupts

Any interrupts required will depend on the specific driver.

4.2 DMA

Any DMA channels required will depend on the specific driver.

4.3 Timers

Any timers required will depend on the specific driver.

4.4 Semaphores

The 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.

4.5 Real-Time Clock

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

4.6 Programmable Flags

Any Programmable Flags required will depend on the specific driver.

4.7 Pins

Any pins required will depend on the specific driver. This will include both the Programmable Flag pins and those required by the peripheral interface of the processor.

5 Supported Features of the Device Driver

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

5.1 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

5.2 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:

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

5.3 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 6.10)
  - `CallbackParameter` – This will always contain the address of the FSS Super Buffer structure. It must remain unaltered within the PID. If the PID does not support peripheral DMA (i.e. returns FALSE in response to the `ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT` command) then this must be passed as the third argument to the Device Manager callback function upon completion of transfer.
  - `ProcessedFlag` - This field is not used in the PID.
  - `pAdditionalInfo` – This field is not used in the PID.

5.4 Command IDs

This section enumerates the commands that are supported/required 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 Device Driver commands that the driver may support. The remaining section describes PID specific commands. Finally, the last subsection describes how to define additional Device 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 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.

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 *
- **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 5.2.
  - Value – ADI_DEV_MODE enumeration

5.4.2 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 may be supported by this class of device driver.

- **ADI_DEV_CMD_GET_INBOUND_DMA_PMAP_ID**
  - Returns the PMAP ID for the device driver's inbound DMA channel.
  - Mandatory for DMA-based PIDs.
  - 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
  - Mandatory for DMA-based PIDs.
  - Value – u32 * (location where the PMAP value is stored)
- **ADI_DEV_CMD_SET_DATAFLOW**
  - Enables/disables dataflow through the device
  - Mandatory.
  - Value – TRUE/FALSE
- **ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT**
  - Determines if the device driver is supported by peripheral DMA
  - Mandatory.
  - Value – u32 * (location where TRUE or FALSE is stored)
- **ADI_DEV_CMD_SET_ERROR_REPORTING**
  - Enables/Disables error reporting from the device driver
  - Optional.
  - Value – TRUE/FALSE
- **ADI_DEV_CMD_GET_MAX_INBOUND_SIZE**
  - Returns the maximum number of data bytes for an inbound buffer
  - Optional.
  - Value – u32 * (location where the size is stored)
- **ADI_DEV_CMD_GET_MAX_OUTBOUND_SIZE**
  - Returns the maximum number of data bytes for an outbound buffer
  - Optional.
  - Value – u32 * (location where the size 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.
  - Mandatory, although the first implementation may simply ignore the command and return a success code regardless.
  - Supported in VisualDSP 5.0 and above
  - 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.

- Mandatory, although the first implementation may simply ignore the command and return a success code regardless.
- Supported in VisualDSP 5.0 and above
- Value – `ADI_DEV_FREQUENCIES` * (new frequencies)

5.4.3 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.

The following commands are mandatory for all types of FSS compliant PID device drivers, and are required for internal use by the FSS.

- **`ADI_FSS_CMD_GET_BACKGRND_XFER_SUPPORT`**
  - Requests the PID to return TRUE or FALSE depending on whether the device supports the transfer of data in the background. For most devices this request equates to whether or not peripheral DMA is supported. For PIDs that need to post-process the results – e.g. for Yellow book decoding – before returning control to the appropriate FSD a value of FALSE should be returned. The value returned will dictate whether the file cache is activated for files opened on the media associated with the PID.
  - Value – Client provided location to store result.
- **`ADI_FSS_CMD_GET_DATA_ELEMENT_WIDTH`**
  - Requests the PID to return the width (in bytes) that defines each data element. This usually equates to the value best supported (or only supported) by the appropriate peripheral DMA. For example, the BF54x ATA interface only supports 16 bit (2 bytes) DMA buffers, whilst the BF54x SD-CARD interface only supports 32 bit (4 bytes) DMA buffers.
  - Value – Client provided location to store result.
- **`ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE`**
  - Requests the 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 PID instance to use the given Heap Index for any dynamically allocated data caches. Whilst its use is not mandatory, the PID must return ADI_FSS_RESULT_SUCCESS even if the command is ignored. The default heap Index for such caches must default 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 PID to return TRUE or FALSE depending on whether the device is to be regarded as Fixed or removable. This command is only applicable when such information is required for formatting purposes.
  - Value – Client provided location to store result.
- **`ADI_PID_CMD_MEDIA_ACTIVATE`**

  - Activates the 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 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 must issue the ADI_FSS_EVENT_MEDIA_REMOVED callback event to the Device Manager Callback function. Please refer to Section 5.4.4 for further details. If the driver detects that media has been inserted it must execute the IDENTIFY or IDENTIFY PACKET command (or whatever is appropriate for the specific driver) and then issue the ADI_FSS_EVENT_MEDIA_INSERTED callback event.
  - Value – NULL.

- **ADI_PID_CMD_DETECT_VOLUMES**
  - Instructs the PID to discover the volumes/partitions available on the media. For each volume detected the PID must issue the ADI_FSS_EVENT_VOLUME_DETECTED event, passing the pointer to the salient volume information as the third argument. refer to Section 5.4.4 for further details.
  - Value – NULL.

- **ADI_PID_CMD_SEND_LBA_REQUEST**
  - Requests the driver 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 PID to take the necessary steps to begin/stop dataflow. This will vary from device to device. Please note that the `{ADI_DEV_CMD_SET_DATAFLOW,(void*)TRUE}` command will not reach the PDD part of a device driver after the first call unless the driver does not support peripheral DMA or a `{ADI_DEV_CMD_SET_DATAFLOW,(void*)FALSE}` command is received in-between, which is not strictly necessary and may even be inefficient to do so, as most drivers are best served by enabling dataflow in the Device Manager once at the beginning of driver initialization.
  - 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.
  - Value – the address of the direct callback function.

The following commands are common for PIDs that support optical media:

- **ADI_PID_CMD_SET_VOLUME_PRIORITY**
  - For PIDs that support optical drives, this command can be used to determine the order of priority for determining which volume type to use. These types are defined in adi_fss.h for Joliet 3, Joliet 2, Joliet 1 and the standard ISO 9660 volumes. Several instances of this command can be used with the type specified by the last command taking highest priority. Upon the command to detect volumes, the PID will attempt to return information for the highest priority on the list, otherwise it will drop down to the next priority and so on until it finds a type it can support. Failing this, it will simply provide support for the standard

ISO9660 volumes with 8.3 file names. The volume type supported is a function of how the optical media was produced.
    - Value - The enumerator value of the volume type to be placed at the top of the list.

- **ADI_PID_CMD_GET_MAX_READ_SPEED**
    - Instructs the PID to supply the maximum read speed capability of the drive in the usual terms of a multiplier on the single speed, i.e. 1x, 2x, …., 32x.
    - Value – Address of a u32 variable to contain the speed multiplier on successful return.
- **ADI_PID_CMD_SET_READ_SPEED**
    - Instructs the PID to adjust the read speed of the drive accordingly.
    - Value – u32 variable containing the speed multiplier.
- **ADI_PID_CMD_GET_MAX_WRITE_SPEED**
    - Instructs the PID to supply the maximum write speed capability of the drive in the usual terms of a multiplier on the single speed, i.e. 1x, 2x, …., 32x.
    - Value – Address of a u32 variable to contain the speed multiplier on successful return.
- **ADI_PID_CMD_SET_WRITE_SPEED**
    - Instructs the PID to adjust the write speed of the drive accordingly.
    - Value – u32 variable containing the speed multiplier.
- **ADI_PID_CDAUDIO_CMD_GET_DISK_INFO**
    - Instructs the PID to supply overall disk information,.
    - Value – Address of an `ADI_PID_CDAUDIO_DISK_INFO` structure to contain the disk information on successful return.
- **ADI_PID_CDAUDIO_CMD_GET_TRACK_INFO**
    - to get information about an individual track
    - Value – Address of an `ADI_PID_CDAUDIO_TRACK_INFO` structure to contain the track information on successful return.

The following commands are common for PIDs that support more than one device on the same data bus:

- **ADI_PID_CMD_SET_NUMBER_DEVICES**
    - For a PID that supports the Parallel ATA interface, this command specifies the number of devices on the chain that this instance of the PID is to interface with.
    - Value – 1 or 2.

The following commands are common for devices that support access to overall device data, and are optional:

- **ADI_PID_CMD_GET_GLOBAL_MEDIA_DEF**
    - Requests the PID to return information regarding the total geometry of the drive.
    - Value – the address of an ADI_FSS_VOLUME_DEF structure to store the overall device information:

      `FileSystemType` - Not used.
      `StartAddress` - The Sector (LBA value) of the first usable sector on the media.
      `VolumeSize` - The total number of sectors on the device.
      `SectorSize` - the size in bytes of each sector.

5.4.4 Defining Device Specific Commands

Additional commands may be defined to cater for the specific requirements of the PID. These commands are only available at configuration time or when the driver is used standalone from the FSS, e.g. for such purposes as partitioning. In some cases these commands may also be applicable to paired FSD – PID drivers.

These commands must be defined in the PID specific header file {pid-ident}.h as follows:

```
enum {
    {NS}_{IFACE}_CMD_START = ADI_PID_CUSTOM_CMD_START,   /* (0x000AA000) */
    {NS}_{IFACE}_CMD_<command-description-1>,
    {NS}_{IFACE}_CMD_<command-description-2>,
    {NS}_{IFACE}_CMD_<command-description-n>,
};
```

5.5 Callback Events

This section enumerates the callback events the PID must generate. 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 (sections 5.4.3, 7) this `void*` parameter must 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 must be directed to the Device Manager provided callback function specified as the last argument, `DMCallback`, passed to the `adi_pdd_Open` function of the PID. It is not necessary to post a deferred callback as the Device Manager callback will do so if a valid DCB queue handle was passed to `adi_dev_Open()`. Support for deferred callbacks is governed upon configuration of the FSS.

The exception 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 PID should 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 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.

- **ADI_DEV_EVENT_BUFFER_PROCESSED**
  Notifies callback function that a chained or sequential I/O buffer has been processed by the device driver. This event is also used to notify that an entire circular buffer has been processed if the driver was directed to generate a callback upon completion of an entire circular buffer.

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.

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.

- **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.
  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`, then the PID can 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.
  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 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 PID. Its values need to be copied in the FSS or application callback prior to returning control to the PID if they are to be retained.

- **ADI_PID_EVENT_DEVICE_INTERRUPT**
  This event is issued in response to the PID handling an interrupt from the device on completion of data transfer. If a device does not provide such an interrupt then this event must be issued upon processing the DMA data transfer complete event, ADI_DEV_EVENT_BUFFER_PROCESSED.
  Value – The address of the Buffer structure associated with the interrupt. This must be the value located in the pBuffer field of the LBA request structure associated with the completion event.

5.6 Return Codes

All API functions of the device driver 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
}
```

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.

5.6.2 FSS Specific Return codes used by a PID driver

The following return codes are defined in the <services/fss/adi_fss.h> header file and are relevant to all PID device drivers:

- **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_DEVICE_IS_LOCKED**
  The device driver is locked as a result of data transfer in progress.
- **ADI_FSS_RESULT_NOT_SUPPORTED**
  The requested operation is not supported by the PID.
- **ADI_FSS_RESULT_SUCCESS**
  General Success.

6 Data structures

6.1 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 should be defined in the PID source module, {pid-ident}.c, and declared as an extern variable in the PID header file, {pid-ident}.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, `__{PID-IDENT}_C__`.
- In the header file, guard the extern declaration:

```
#if !defined(__{PID-IDENT}_C__)
extern ADI_DEV_PDD_ENTRY_POINT {PID_IDENT}_EntryPoint;
:
#endif
```

6.2 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 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 );`

A default table should be declared in the PID header file, {pid-ident}.h, and guarded against inclusion in the PID Source module, and should only only be to included in an application module if the developer defines the macro, `_{PID-IDENT}_DEFAULT_DEF_`:

```
#if !defined(__{PID-IDENT}_C__)
:
#if defined(_{PID-IDENT}_DEFAULT_DEF_)
static ADI_DEV_CMD_VALUE_PAIR {PID-IDENT}_ConfigurationTable[] = { … };
:
#endif
:
#endif
```

6.3 Device Definition Structure, ADI_FSS_DEVICE_DEF

This structure is used to instruct the FSS how to open and configure the 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()`. For most PIDs this value will be `0`.
`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 a PID its value should be assigned to `&{PID_IDENT}_EntryPoint`.
`pConfigTable`	This is a pointer to the table of command-value pairs to configure the PID, and its value should be assigned to `{PID_IDENT}_ ConfigurationTable`.
`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 most PIDs this value will be `ADI_DEV_DIRECTION_BIDIRECTIONAL`.
`DeviceHandle`	This is the location used for internal use to store the Device Driver Handle returned 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 this PID. This is particularly useful for the PID of a device with removable media, so that the same letter is used on replacement of media.

A default instantiation of this structure is declared in the PID header file, {pid-ident}.h, and guarded against inclusion in the PID Source module, and should only only be to included in an application module if the developer defines the macro, `_{PID-IDENT}_DEFAULT_DEF_`:

```
#if !defined(__{PID-IDENT}_C__)
:
#if defined(_{PID-IDENT}_DEFAULT_DEF_)
static ADI_FSS_DEVICE_DEF {PID-IDENT}_Def = { … };
:
#endif
:
#endif
```

6.4 Volume Definition Structure, ADI_FSS_VOLUME_DEF

This structure is used within the 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, e.g. ADI_FSS_FSD_TYPE_CDDATA_MODE1.
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 section.
DeviceNumber	This is used to indicate the device number on a chain of devices where relevant. For example, a Parallel ATA driver could support a Master (DeviceNumber=0) and Slave (DeviceNumber=1) devices on a single chain. If this is not relevant to the PID, it should assign a value of 0.

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

6.5 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 an instantiation of this should be send to the PID with either an ADI_PID_CMD_SEND_LBA_READ_REQUEST or ADI_PID_CMD_SEND_LBA_WRITE_REQUEST command prior to enabling dataflow in the PID. 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 can be ignored for PIDs where this is not relevant.
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.

6.6 CD AUDIO Disk Information Structure, ADI_FSS_CDAUDIO_DISK_INFO

This structure is used to pass CD AUDIO disk information to an appropriate FSD, in response to the ADI_PID_CDAUDIO_CMD_GET_DISK_INFO command. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
```

```
    u32             DeviceNumber;
    u32             Tracks;
} ADI_FSS_CDAUDIO_DISK_INFO;
```

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

DeviceNumber	The Device Number on the chain. This can be ignored for PIDs where this is not relevant.
Tracks	The PID must set this field with the number of tracks found on the audio CD.

Use of this structure is only required for PIDs that connect to optical devices.

6.7 CD AUDIO Track Information Structure, ADI_FSS_CDAUDIO_TRACK_INFO

This structure is used to pass CD AUDIO track information to an appropriate FSD, in response to the ADI_PID_CDAUDIO_CMD_GET_TRACK_INFO command. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
    u32             DeviceNumber;
    u32             Index;
    u32             Address;
    u32             Size;
} ADI_FSS_CDAUDIO_TRACK_INFO;
```

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

DeviceNumber	The Device Number on the chain. This can be ignored for PIDs where this is not relevant.
Index	The index number of the track in the range 1 – number of tracks as returned in the ADI_FSS_CDAUDIO_DISK_INFO structure. This is set by the FSD before the command is sent.
Address	The starting point of the track in LBA sector format.
Size	The number of sectors in the track.

Use of this structure is only required for PIDs that connect to optical devices.

6.8 CD AUDIO CDTEXT Information structure, ADI_FSS_CDAUDIO_CD_TEXT

This structure is used to pass raw CDTEXT information to an appropriate FSD, in response to the ADI_PID_CDAUDIO_CMD_GET_CD_TEXT command. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
    u32             DeviceNumber;
    u32             Size;
    u32             Read;
    u8              *pData;
} ADI_FSS_CDAUDIO_CD_TEXT;
```

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

DeviceNumber	The Device Number on the chain. This can be ignored for PIDs where this is not relevant.
Size	The maximum number of bytes to be read from the CD.
Read	The PID is to set this value to indicate the number of bytes read. This may be

	less than or equal to the number specified in the `Size` field.
`pData`	The raw CDTEXT data is to be copied to this buffer.

Use of this structure is only required for PIDs that connect to optical devices.

6.9 Use of the Device Driver ADI_DEV_1D_BUFFER structure

The only Device Driver buffer structure that can be used with a PID is the `ADI_DEV_1D_BUFFER`. For a detailed description of its definition, please refer to the System Services and Device Driver Manual. When passed to the PID via a call to adi_dev_Read/Write, the contents of these structures are relevant for PIDs that do not support peripheral DMA. For Internal data transfers, e.g. SCSI packets etc., the values will also need to be set accordingly. Within this context, the data fields take on the following meanings:

`Data`	The address of the buffer to fill/empty. This can be assumed to be of size, `ElementCount*ElementWidth`
`ElementCount`	The number of words to transfer.
`ElementWidth`	The size in bytes of each word, in bytes. This should be the same as returned in response to the `ADI_FSS_CMD_GET_DATA_ELEMENT_WIDTH` command.
`CallbackParameter`	The address of the enclosing `ADI_DEV_1D_BUFFER` structure. It should be passed as the third argument in a callback to the Device Manager callback function upon completion of data transfer.
`pAdditionalInfo`	Not used.
`ProcessedElementCount`	This should be set to the number of elements transferred, either in the Read/Write function of the driver or when called from the FSS callback function.
`ProcessedFlag`	Not used.

Please refer to the section on callbacks for further information

6.10 Use of the ADI_FSS_SUPER_BUFFER structure

Whilst calls to adi_dev_Read/Write pass the address of an `ADI_DEV_BUFFER` structure, within the Framework of the FSS, the address actually points to the address of both the `ADI_DEV_BUFFER` structure and an envelope structure of type `ADI_FSS_SUPER_BUFFER`. This structure has the `ADI_DEV_BUFFER` structure as its first member and contains other important information required for communication between the other components of the File System Service framework. This *Super Buffer* has the following data fields:

`Buffer`	The `ADI_DEV_BUFFER` structure required for the transfer. Please note that this is not a pointer field.
`pBlock`	Used in the FSS File Cache. Its value must remain unchanged by the PID. For internal PID transfers it must to set to NULL.
`LastinProcessFlag`	Used in the FSS File Cache. Its value must remain unchanged by the PID. For internal PID transfers it must to set to FALSE.
`LBARequest`	The ADI_FSS_LBA_REQUEST structure for the associated buffer. If the buffer forms part of a chain and the SectorCount value is zero, then the LBA request of

	a previous sub buffer covers the data in this sub buffer also.
SemaphoreHandle	The Handle of the Semaphore to be posted upon completion of data transfer. For internal PID transfers it must to set to the semaphore associated with the PID. For transfers initiated externally to the PID its value must remain unchanged by the PID.
pFileDesc	Used in the FSS File Cache. Its value must remain unchanged by the PID. For internal PID transfers it must to set to NULL.
FSDCallbackFunction	Set by an FSD. Its value must remain unchanged by the PID. For internal PID transfers it must to set to NULL.
FSDCallbackHandle	Set by an FSD. Its value must remain unchanged by the PID. For internal PID transfers it must to set to NULL.
PIDCallbackFunction	The address of the ADI_DCB_CALLBACK_FN function that is to be called by the FSS upon receipt of the ADI_DEV_EVENT_BUFFER_PROCESSED and ADI_PID_EVENT_DEVICE_INTERRUPT events. Upon receipt of the ADI_PID_CMD_SEND_LBA_REQUEST command, the PID must use the pBuffer value in the LBA request structure to access the super buffer and set this value accordingly. Similarly its value must be set for internal PID transfers.
PIDCallbackHandle	The address of the PID Device Driver Instance. Please note, this must the instance data of the PDD section of the device driver and not its *Device Handle*. Upon receipt of the ADI_PID_CMD_SEND_LBA_REQUEST command, the PID must use the pBuffer value in the LBA request structure to access the super buffer and set this value accordingly. Similarly its value must be set for internal PID transfers.

7 Initialization

7.1 Description

The File System Service (FSS) will automatically open and configure a PID by issuing calls to adi_dev_Open() and adi_dev_Control() according to its entry in the Devices' Linked list.

To use a PID within the FSS define an `ADI_FSS_DEVICE_DEF` structure (Section 6.3) and set the `pEntryPoint` field to the address of the Entry point structure (Section 6.1) and the `pConfigTable` field to the address of a configuration table of command-value pairs (Section 6.2).

Please note that the default structures can be used provided that the `_{PID-IDENT}_DEFAULT_DEF_` macro is defined ahead of the inclusion of the driver header file:

```
#define _{PID-IDENT}_DEFAULT_DEF_
#include <{pid-ident.h}>
```

In which case the `ADI_FSS_DEVICE_DEF` structure will be given the name, `{PID-IDENT}_Def`.

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". A default value should not be specified in the default definition.

The address of the `ADI_FSS_DEVICE_DEF` structure can then be registered with the File System Service, using the following command-value pair in the configuration table passed to `adi_fss_Init()`, e.g.:

```
{ ADI_FSS_CMD_ADD_DRIVER,(void*)&{PID-IDENT}_Def },
```

Once opened and configured, the FSS will request media activation with the command-value pair:

```
{ ADI_PID_CMD_MEDIA_ACTIVATE, (void *)TRUE },
```

At which point, The PID should initialize the Media Device and return `ADI_FSS_RESULT_SUCCESS` if successful and `ADI_FSS_RESULT_FAILED` otherwise. If the PID supports peripheral DMA, the PID should initiate data flow with the Device Manager at this stage with the following command pair:

```
{ ADI_DEV_CMD_SET_DATAFLOW, (void*)TRUE },
```

From this point the DMA controller will respond to requests from the mass storage device as and when required. The handle to pass to `adi_dev_Control()` is that as the third argument passed to `adi_pdd_Open()`.

Once a PID has been activated, it is polled to determine whether media is present. Before doing so, the FSS will pass the address of its callback function to the PID with the following command-value pair:

```
{ ADI_PID_CMD_SET_DIRECT_CALLBACK, (void *)<FSS-call-back-function> },
```

This callback function is to be called upon detection of media insertion/removal and also upon detection of a usable partition or volume.

To poll for the presence of media, the FSS sends the following command-value pair to the PID:

```
{ ADI_PID_CMD_POLL_MEDIA_CHANGE, (void*)NULL },
```

If media is detected the PID must issue a callback event to the FSS callback function as defined above. This callback must be live and be made with the following arguments:

1. The address of the `DeviceHandle` argument, supplied as the third argument passed to `adi_pdd_Open()`.
2. The `ADI_FSS_EVENT_MEDIA_INSERTED` event.
3. The address of a u32 variable. The contents of this variable should be set 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.

If the PID detects the removal of media, it must issue the `ADI_FSS_EVENT_MEDIA_REMOVED` callback event to the `DMcallback` function. The arguments are:

1. The address of the `DeviceHandle` argument, supplied as the third argument passed to `adi_pdd_Open()`. and the third callback argument must be
2. The `ADI_FSS_EVENT_MEDIA_REMOVED` event.
3. The address of a u32 variable. The contents of this variable should be set to the Device Number of the device on the chain for which media is detected, if appropriate. Its value can be ignored on return.

In response to media insertion the FSS will instruct the PID to detect usable volumes/partitions with the following command-value pair:

```
{ ADI_PID_CMD_DETECT_VOLUMES, (void*)<device-number> },
```

The `<device-number>` will indicate, where applicable, the device number of the physical device on the chain.

Upon detection of a valid volume, the PID must issue another `ADI_FSS_EVENT_VOLUME_DETECTED` live callback to the FSS callback function. The arguments are:

1. The address of the `DeviceHandle` argument, supplied as the third argument passed to `adi_pdd_Open()`. and the third callback argument must be
2. The `ADI_FSS_EVENT_VOLUME_DETECTED` event.
3. The address of an `ADI_FSS_VOLUME_DEF` structure defining the volume. Please refer to Section 6.4 for details about the definition and assignment of this structure.

7.2 Initialization when used standalone

In cases where the PID is to be used directly, 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 PID separate from the context of the File System Service. This section details what is required.

The device driver definition structure, `{PID-IDENT}_Def`, (Section 6.3) provides most of the requirements for the call to `adi_dev_Open()` to open the PID device driver:

```
Result = adi_dev_Open(
                          <DeviceManagerHandle>,
                          {PID-IDENT}_Def.pEntryPoint,
                          {PID-IDENT}_Def.DeviceNumber,
                          &{PID-IDENT}_Def.DeviceHandle,
                          &{PID-IDENT}_Def.DeviceHandle,
                          {PID-IDENT}_Def.Direction,
                          <DMAManagerHandle>,
                          <DCBQueueHandle>,
                          <Callback-function>
                      );
```

The other arguments need to be supplied. The `<DeviceManagerHandle>` and `<DMAManagerHandle>` are what are 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 PID are to be deferred.

Next, the PID needs to be configured with the required configuration settings (Section 7.4) and any optional settings required, for instance to over-ride the defaults settings (Section 7.3).

The `<Callback-function>` passed in the call to `adi_dev_Open()` will be called, not from the PDD section of the device driver, 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 callbacks 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 7.1.

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 `&{PID-IDENT}_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 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 6.4 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 data transfer completion event in terms of the peripheral DMA. The third argument, `pArg`, in this case is the `CallbackParameter` field

of the ADI_DEV_1D_BUFFER structure (Section 6.9) passed via the corresponding call to adi_dev_Read() or adi_dev_Write(). This value should be address of the enclosing ADI_FSS_SUPER_BUFFER structure. In this event the callback function must call the PID via its callback function identified by the PIDCallbackFunction member of the ADI_FSS_SUPER_BUFFER structure. The first argument in this call must be set to the value of the PIDCallbackHandle member of the ADI_FSS_SUPER_BUFFER structure and the Event and pArg arguments simply passed on:

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

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.

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

7.3 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.

<This table should list the default settings for the device driver. The first column should contain the description of the setting, the next column the default value, next is the list of possible values and the last column contains the command ID that can be sent to override the default setting. For example, a video driver may default to NTSC formatted video but may also support PAL formatted video. Add entries to this table as appropriate.>

Item	Default Value	Possible Values	Command ID
Pend Argument	NULL	The address of variable or structure to be passed to the FSS pend function where applicable.	ADI_FSS_CMD_SET_PEND_ARGUMENT
Cache Heap Index	-1	The heap index to use for the allocation of data tansfer buffers.	ADI_FSS_CMD_SET_CACHE_HEAP_ID
Number of devices on chain	1	1 or 2.	ADI_PID_CMD_SET_NUMBER_DEVICES
Read Speed (optical devices)	As device dictates	1 to *n* where *n* is the maximum read speed of the device.	ADI_PID_CMD_SET_READ_SPEED

Table 3 - Default Settings

7.4 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.

<This table should list all configuration items that the application is required to specify typically before dataflow on the driver can be enabled. For example, the command to set the dataflow method is required by the Device Manager before the client can enable dataflow. (This command is already included in the table and should not be deleted. Entries in the table should be included for other settings such as which timers or programmable flag pins the driver is to use or, in the case of a device driver leveraging the services of a lower level driver, which lower level driver to use. For example, an audio codec device driver may use a SPORT driver for data transport. If for some reason it was inappropriate for the device driver to default to a specific SPORT device, the device driver may require that the application specify which SPORT device is to be used before enabling dataflow.>

Item	Possible Values	Command ID
Dataflow method	See section 5.2	`ADI_DEV_CMD_SET_DATAFLOW_METHOD`
FSS Callback	The address of the FSS function.	`ADI_PID_CMD_SET_DIRECT_CALLBACK`
Data Element Width	See section 5.4.3.	`ADI_FSS_CMD_GET_DATA_ELEMENT_WIDTH`
Backround Transfer Support	See section 5.4.3.	`ADI_FSS_CMD_GET_BACKGRND_XFER_SUPPORT`

Table 4 – Additional Required Settings

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 most – if not all – devices are concerned, data transfer is active from the receipt of the command to transfer a number of sectors and the completion of transfer. We will refer to this as a *DRQ block*, after the process associated with ATA devices. 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`.

Where a chain of such buffers defines contiguous data on the media, and a single LBA request has been sent to the PID, to cover the chain or parts thereof, there will be several *host* transfer completion events to the one *device* transfer completion event, or DRQ block. It will be necessary for the PID to lock access to the driver for the duration of the DRQ block. This is achieved by maintaining a Lock Semaphore, which should be created as a binary semaphore with an initial count of one using the Semaphore Service, e.g.

```
adi_sem_Create ( 1,  &pDevice->LockSemaphoreHandle, NULL );
```

this semaphore must be pended on upon receipt of the `ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE` command:

```
adi_sem_Pend ( pDevice->LockSemaphoreHandle, ADI_SEM_TIMEOUT_FOREVER);
```

and posted on receipt of the `ADI_FSS_CMD_RELEASE_LOCK_SEMAPHORE` command:

```
adi_sem_Post ( pDevice->LockSemaphoreHandle);
```

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

1. Acquire Lock Semaphore from the 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 PID by passing the command-value pair, e.g.:

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

   The PID should assign the `PIDCallbackFunction` and `PIDCallbackHandle` (section 6.10) fields of the `ADI_FSS_SUPER_BUFFER` structure pointed to by the `pBuffer` field of the LBA Request structure and store a copy of the LBA request structure in its instance data.

3. Then the FSD submits the buffer chain to the 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 PID

   ```
   { ADI_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.

Upon a *host* transfer completion event, a PID that supports peripheral DMA[1] will automatically issue the `ADI_DEV_EVENT_BUFFER_PROCESSED` event via the Device Manager part of the device driver. PIDs that do not support peripheral DMA must issue this event on completion of each sub-buffer in the chain. Upon completion of each DRQ unit the PID must issue the `ADI_PID_EVENT_DEVICE_INTERRUPT` event. These callbacks must be made via the Device Manger with the following arguments:

[1] i.e. responds with `TRUE` to the `ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT` command.

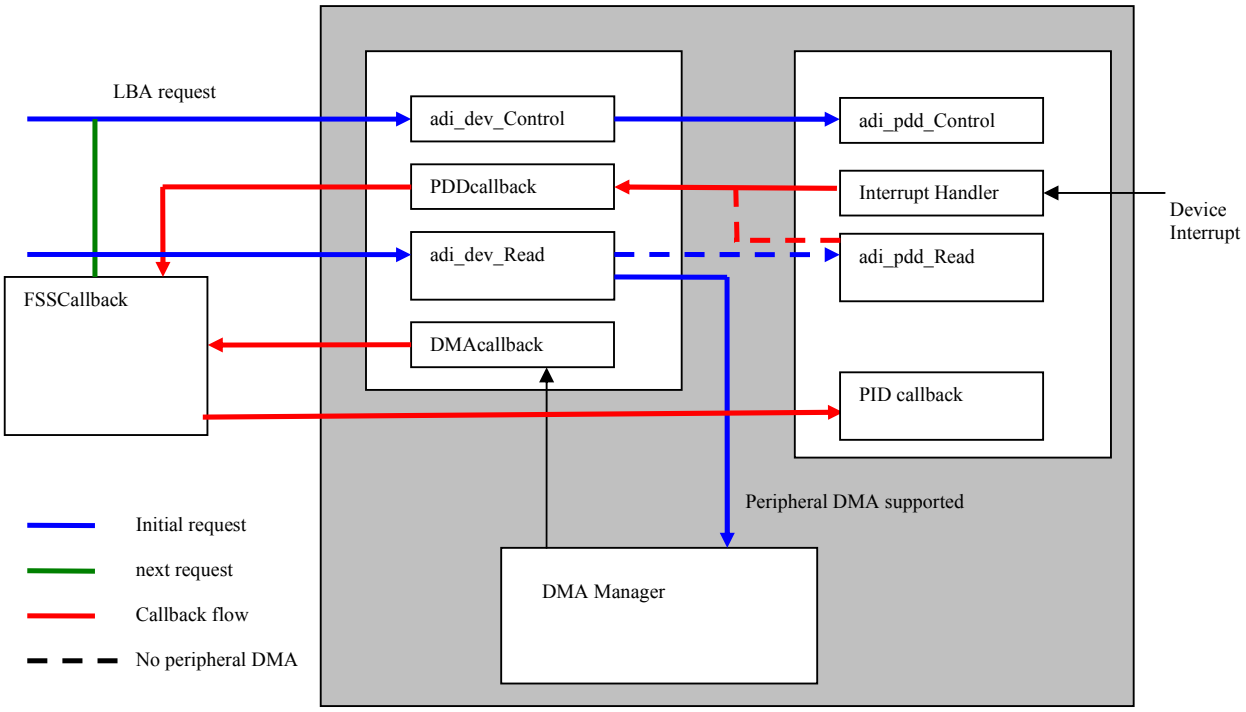
1. The `DeviceHandle` argument, supplied as the third argument passed to `adi_pdd_Open()`.
2. The appropriate event code.
3. The address of `pBuffer` value in the LBA request structure saved at stage 2 in the submission process above.

In reply to these events, the FSS will make a call into the PID using the `PIDCallbackFunction` and `PIDCallbackHandle` (section 6.10) fields of the `ADI_FSS_SUPER_BUFFER` structure:

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

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

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



9 Hardware Considerations

<This section should document any specific hardware considerations for the driver. For example if pins to the device define the address of the device on a bus, this section should describe how this relates to the configuration settings for the device driver. For example an ADV7171 video encoder uses a pin to set bit 1

of its TWI slave address register. Typically a command ID would exist that tells the driver the TWI slave address. Items like that should be discussed here.>