ANALOG DEVICES

SECURE DIGITAL HOST (ADI_SDH) PHYSICAL INTERFACE DRIVER DOCUMENT

DATE: 26 OCTOBER 2007

Table of Contents

Table of Figures

Document Revision History

Date	Description of Changes
22 June, 2007	Initial Release
20 July 2007	Replaced ‘mutex’ with ‘lock semaphore’ throughout
26 Oct, 2007	Added support to Version 2.0 Standard capacity SD and Higher Capacity SD (SDHC) cards

Table 1 - Revision History

1 Overview

This document describes the functionality of Secure Digital Host Physical Interface Driver (to be termed as SDH PID) that adheres to Analog Devices' File System Service (FSS) Model. SDH is available on ADSP-BF54x family processors. The driver supports accessing memory sections of Version 1.X and Version 2.0 Standard capacity Secure Digital (SD) cards, Higher Capacity Secure Digital (SDHC) cards and Secure Digital IO cards with memory (SDIO-Combo cards). Support to Multimedia cards (MMC) and I/O functions of SDIO and SDIO-combo cards will be available in future.

2 Files

The files listed below comprise the SDH PID 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/sdh/adi_sdh.h`**
  This file contains all definitions, function prototypes etc. for the SDH Driver.

- **`<drivers/pid/adi_ata.h`**
  This file contains definitions of ATA Partition Table Structure and lists supported Partition type IDs.

- **`<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/pid/sdh/adi_sdh.c>`**
  This file contains all the source code for the SDH Device Driver. All source code is written in ‘C’. There are no assembly level functions in this driver.

- **`<Blackfin/lib/src/drivers/pid/sdh/adi_sdh_reg.h>`**
  This file contains definitions of all MMR addresses and the bitfield access macros specific to SDH.

3 Lower Level Drivers

SDH PID does not use any lower level device drivers.

4 Resources Required

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

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

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. SDH driver uses *general* heap defined in the FSS. Application can set SDH *cache* heap index by using following command-value pair, which can be sent as a part of default SDH configuration table in SDH PID definition structure.

```
{ 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

SDH PID requires memory for three interrupt handlers – SDH DMA data interrupt handler, SDH DMA error interrupt handler and SDH status interrupt handler. Each interrupt handler requires memory of size `ADI_INT_SECONDARY_MEMORY`.

4.2 DMA

SDH only supports SD-protocol (half-duplex) and requires/uses one DMA channel to transfer data between the processor and connected memory device. Application should allocate memory for one SDH DMA channel of size `ADI_DMA_CHANNEL_MEMORY` during the initialization process.

4.3 Timers

SDH PID does not require or use timer service.

4.4 Semaphores

The SDH PID requires two semaphores, one for a Lock Semaphore to maintain exclusive access to the SDH driver from one process at a time, and one for signalling completion of internal data transfers. The Semaphore Service must be used to create and manipulate all semaphores. Each semaphore requires memory of size `ADI_SEM_SEMAPHORE_MEMORY`.

4.5 Real-Time Clock

Use of the RTC Service is not required by SDH PID.

4.6 Programmable Flags

SDH PID does not require or use programmable flags.

4.7 Pins

SDH requires a minimum of three pins to connect to Command, Clock and Data lines of the SD Slot. Application can choose to operate SDH in ‘wide-bus’ mode, which expands the SDH data bus to 4-bit wide, which requires 3 more pins. On processors where pin multiplexing is used, depending on the SDH register values, the SDH PID automatically reconfigures programmable flag (PF) pins to enable required pins for its use. Applicant must use caution to insure that the SDH does not use any Programmable Flag pins used by any other general purpose I/O device, and vice-versa.

5 Supported Features of the Device Driver

This section describes what features are supported by SDH PID.

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_OUTBOUND	Supports transmission of data 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

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

```
{ 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 (refer to FSS documentation for FSS super buffer structure)
  - `CallbackParameter` – This will always contain the address of the FSS Super Buffer structure and must remain unaltered.
  - `ProcessedFlag` - This field is not used in the SDH driver.
  - `pAdditionalInfo` – This field is not used in the SDH driver.

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 supported by SDH PID. The third section describes common PID commands supported by SDH PID and the remaining section describes commands specific to SDH 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.

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 are supported by SDH PID.

- **ADI_DEV_CMD_SET_DATAFLOW**
  - Enables/disables dataflow through the device
  - Value – TRUE/FALSE, TRUE to enable dataflow
  - Default – Dataflow disabled
- **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)
  - SDH is a half-duplex device and is mapped to a single DMA channel. To have a better control over DMA dataflow, the SDH DMA and buffers are handled by the driver itself rather than the Device manager. So the driver will return FALSE as command value.
- **ADI_DEV_CMD_FREQUENCY_CHANGE_PROLOG**
  - Notifies device driver immediately prior to a CCLK/SCLK frequency change. If the SCLK frequency increases then the SDH PID timings are adjusted prior to frequency change.
  - 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 SDH PID timings are adjusted subsequent to frequency change.
  - Value – `ADI_DEV_FREQUENCIES` * (new frequencies)

5.4.3 Common PID Commands

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

- **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.
  - Value – Client provided location to store result.
  - SDH PID supports background data transfer and hence returns TRUE as command value
- **ADI_FSS_CMD_GET_DATA_ELEMENT_WIDTH**
  - Requests the PID to return the width (in bytes) that defines each data element.
  - Value – Client provided location to store result.
  - SDH DMA only supports 32-bit (4-byte) data transfers.
- **ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE**
  - Requests the SDH 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 SDH 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 SDH PID instance to use the given Heap Index for any dynamically allocated data caches.
  - 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.
  - Value – Client provided location to store result.
  - SDH PID only supports removable devices and returns TRUE as command value
- **ADI_PID_CMD_MEDIA_ACTIVATE**
  - Activates the SDH device, configuring it for use.
  - Value – TRUE to activate, FALSE to deactivate
- **ADI_PID_CMD_POLL_MEDIA_CHANGE**
  - Instructs the SDH PID to check the status of the device for the removal or insertion of media. IF the PID detects that a media (card) has been inserted, it performs a card identification process and issues ADI_FSS_EVENT_MEDIA_INSERTED callback event to FSS notifying a new media insertion.
  - Value – NULL.
- **ADI_PID_CMD_DETECT_VOLUMES**
  - Instructs the SDH PID to discover the volumes/partitions available on the media. For each volume detected the SDH PID issues the ADI_FSS_EVENT_VOLUME_DETECTED event to FSS, passing the pointer to the salient volume information as the third argument.
  - Value – NULL.
- **ADI_PID_CMD_SEND_LBA_REQUEST**
  - Requests the SDH 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 SDH PID to take the necessary steps to begin/stop dataflow.
  - 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.
- **ADI_PID_CMD_GET_GLOBAL_MEDIA_DEF**
  - Requests the SDH PID to return information regarding the total geometry of the inserted device.
  - 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 SDH PID Specific Commands

- **`ADI_SDH_CMD_SET_SUPPORTED_CARD_OPERATING_CONDITION`**
  - Sets the range of card operating condition (voltage) supported by the hardware.
  - Value – u32.
  - Default driver settings – Operating conditions supported by ADSP–BF548 Ez-Kit rev 1.1

  Following table lists the range of operating voltages supported by Version 1.X / Version 2.0 Standard capacity SD, Higher Capacity SD (SDHC) and SDIO cards. Applicant must cross-check this table with the operating voltage window supported by their hardware and pass an u32 type value with bits corresponding to the supported operating voltage ranges marked as 1.

u32 Value Bit location	Operating voltage window
0 to 3	Reserved – must be set to 0
4	1.6 to 1.7
5	1.7 to 1.8
6	1.8 to 1.9
7	1.9 to 2.0
8	2.0 to 2.1
9	2.1 to 2.2
10	2.2 to 2.3
11	2.3 to 2.4
12	2.4 to 2.5
13	2.5 to 2.6
14	2.6 to 2.7
15	2.7 to 2.8
16	2.8 to 2.9
17	2.9 to 3.0
18	3.0 to 3.1
19	3.1 to 3.2
20	3.2 to 3.3
21	3.3 to 3.4
22	3.4 to 3.5
23	3.5 to 3.6
24 to 31	Reserved – must be set to 0

Table 3 – Card Operating conditions

- **`ADI_SDH_CMD_SET_CARD_LOCK_ENABLED`**
  - Applicant must use this command to pass the card lock (mechanical switch) status of the inserted media (card) to SDH PID. The position of this switch is unknown to the internal circuitry of the inserted media, and hence the SDH driver cannot detect it automatically. It is applicant's responsibility to monitor this switch using a port flag pin and inform the switch status to SDH driver. When the switch is in lock status, the SDH driver write-protects the media by preventing all write operations to it.
  - Value – TRUE/FALSE. (TRUE when the card is locked)
  - Default driver settings – assumes the card is not locked

- **ADI_SDH_CMD_ENABLE_WIDEBUS**
  - Enables/Disables 'wide-bus' mode. SDH can be configured to use 1-bit or 4-bit data bus to exchange data between the processor and inserted media. 4-bit data bus (termed as wide-bus mode) significantly reduces the memory access time and it is recommended to enable this mode whenever supported. Some card types (like Multimedia and low-speed SDIO cards) only support 1-bit data bus and the SDH PID will bypass the wide-bus settings while accessing such media. Please note that wide-bus mode will increase the number of pins required for SDH access from 3 to 6.
  - Value – TRUE/FALSE. (TRUE to enable wide-bus mode, FALSE to disable it)
  - Default driver settings – Enables wide-bus mode.

- **SD I/O specific commands, Commands to set/clear SD card password etc.,**
  - To be added in future

5.5 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
}
```

5.5.1 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.

5.5.2 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 Point

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

- ADI_SDH_EntryPoint

6.2 Device Definition Structure, ADI_FSS_DEVICE_DEF

This structure is used to instruct the FSS how to open and configure the SDH 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 SDH 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()`. Refer section 6.1 for SDH PID entry point
`pConfigTable`	This is a pointer to the table of command-value pairs to configure the SDH PID
`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()`. Refer section 5.1 for list of directionalities supported by SDH PID
`DeviceHandle`	This is the location used for internal use to store the SDH PID 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 SDH 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 SDH PID header file, adi_sdh.h, and guarded against inclusion in the PID Source module, and should only be to included in an application module if the developer defines the macro, `_ADI_SDH_DEFAULT_DEF_`

6.3 Volume Definition Structure, ADI_FSS_VOLUME_DEF

This structure is used within the SDH 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_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 section.
DeviceNumber	This is used to indicate the device number on a chain of devices. SDH PID always sets this value to 0.

The FSS regards this structure as volatile and makes a copy of its contents.

6.4 LBA Request, ADI_FSS_LBA_REQUEST

This structure is used to pass a request for a number of sectors to be read from the inserted media. The address of an instantiation of this should be send to the SDH PID with ADI_PID_CMD_SEND_LBA_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	Value ignored by SDH 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.

6.5 Use of the Device Driver ADI_DEV_1D_BUFFER structure

The only Device Driver buffer structure that can be used with a SDH PID is the ADI_DEV_1D_BUFFER. For a detailed description of its definition, please refer to the System Services and Device Driver Manual.

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 will be set to the number of elements transferred from/to the SDH device. If the data transfer fails abnormally, the SDH PID interrupt handler notifies the failure to the applicant by setting this field to 0.
ProcessedFlag	Not used.

6.6 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 and the SDH PID only supports ADI_DEV_1D_BUFFER type buffers
pBlock	Used in the FSS File Cache.
LastinProcessFlag	Used in the FSS File Cache.
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.
pFileDesc	Used in the FSS File Cache.
FSDCallbackFunction	Set by an FSD.
FSDCallbackHandle	Set by an FSD.
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.
PIDCallbackHandle	The address of the SDH PID Device Driver Instance.

7 Initialization

7.1 Description

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

To use SDH PID within the FSS, define an `ADI_FSS_DEVICE_DEF` structure (Section 6.2). Please note that the default structure can be used provided that the `_ADI_SDH_DEFAULT_DEF_` macro is defined ahead of the inclusion of the driver header file, in which case the `ADI_FSS_DEVICE_DEF` structure will be given the name, `ADI_SDH_Def`.

Please note that the FSS will endeavour 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 is not be specified in the default definition - `ADI_SDH_Def`.

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*)&ADI_SDH_Def },
```

Once SDH PID is opened and configured, the FSS requests media activation with the command-value pair:

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

On receiving the above command-value pair, the SDH PID initializes the SDH registers and return `ADI_FSS_RESULT_SUCCESS` if successful and `ADI_FSS_RESULT_FAILED` otherwise. At this stage, the SDH PID automatically initiates its data flow with the Device Manager with the following command-value 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. Once SDH PID has been activated, it is polled to determine whether media is present in the SD slot. Before doing so, the FSS passes 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 will 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 SDH PID:

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

If a valid media is detected, the SDH PID issues a callback event to the FSS callback function as defined above. This is a live callback and 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. SDH PID sets the variable value to 0 (indicating the Device Number). 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 SDH PID detects the removal of media, it issues 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. SDH PID sets the variable value to 0 (indicating the Device Number.

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 SDH PID rejects the <device-number> assumes it to be 0.

Upon detection of a valid volume, the PID issues another ADI_FSS_EVENT_VOLUME_DETECTED live callback event 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 6.3 for details about the definition and assignment of this structure.

7.2 Initialization when used standalone

In cases where the SDH 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, `ADI_SDH_Def`, (Section 6.2) provides most of the requirements for the call to `adi_dev_Open()` to open the PID device driver:

```
Result = adi_dev_Open(
                          <DeviceManagerHandle>,
                          ADI_SDH_Def.pEntryPoint,
                          ADI_SDH_Def.DeviceNumber,
                          &ADI_SDH_Def.DeviceHandle,
                          &ADI_SDH_Def.DeviceHandle,
                          ADI_SDH_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 SDH PID are to be deferred.

Next, the PID needs to be configured with the required configuration settings (Section 7.4), 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 SDH 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 SDH PID Device Handle, which will be the same value as given in the call to `adi_dev_Open()`, namely `&ADH_SDH_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 6.3 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 6.5) 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 SDH 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.

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
Supported operating conditions	0x00FF8000	Hardware specific	ADI_SDH_CMD_SET_SUPPORTED_CARD_OPERATING_CONDITION
Card Lock Mechanical switch status	FALSE	TRUE/FALSE	ADI_SDH_CMD_SET_CARD_LOCK_ENABLED
Enable Wide-Bus mode	TRUE	TRUE/FALSE	ADI_SDH_CMD_ENABLE_WIDEBUS

Table 4 - 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.

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 5 – 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 SDH is 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*. 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 SDH 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 is necessary for the SDH PID to lock access to the driver for the duration of the DRQ block. This is achieved by maintaining a Lock Semaphore, which is a binary semaphore with an initial count of one using the Semaphore Service, e.g.

This semaphore will 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 SDH 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 SDH PID by passing the command-value pair, e.g.:

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

   The SDH PID assigns the `PIDCallbackFunction` and `PIDCallbackHandle` (section 6.6) 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 SDH 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 SDH 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, for SDH PID, the `ADI_DEV_EVENT_BUFFER_PROCESSED` event will be issued via the Device Manager part of the device driver. Upon completion of each DRQ unit the SDH PID issues the `ADI_PID_EVENT_DEVICE_INTERRUPT` event. These callbacks must be made via the Device Manger 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 saved at stage 2 in the submission process above.

In reply to these events, the FSS will make a call into the SDH PID using the `PIDCallbackFunction` and `PIDCallbackHandle` (section 6.6) fields of the `ADI_FSS_SUPER_BUFFER` structure:

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

In this function the SDH PID checks for data transfer completion status. If the data transfer for this DRQ block fails, the `ProcessedElementCount` of the buffer corresponding to the DRQ block will be set to 0.. Furthermore, in response to the `ADI_PID_EVENT_DEVICE_INTERRUPT` event, the SDH PID releases the PID Lock Semaphore and post the PID Semaphore *only* if the `SemaphoreHandle` value of the `ADI_FSS_SUPER_BUFFER` equals that of the SDH 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.

/********* Diagram to be added in future **********/

9 Hardware Considerations

On processors where pin multiplexing is used, depending on the SDH register values, the SDH PID automatically reconfigures programmable flag (PF) pins to enable required pins for its use. Applicant must use caution to insure that the SDH does not use any Programmable Flag pins used by any other general purpose I/O device, and vice-versa.