ANALOG DEVICES

GENERIC FSD DESIGN DOCUMENT

20 JULY 2007

Table of Contents

Table of Figures

Document Revision History

Date	Description of Changes
31 May 2006	Initial Draft
6 June 2006	Minor typographical changes
9 June 2006	Amended Lower level driver section
20 June 2006	Added file open mode requirements.
11 July 2006	Updated draft
23 August 06	Amended to reflect decision re command code names and changed functionality concerning the retrieval of block size information.
1 November 2006	Amended control commands and updated configuration details
27 March 2007	Updated for revised commands
13 June 2007	Revised framework
20 July 2007	Replaced ‘mutex’ with ‘lock semaphore’ throughout.

Table 1 - Revision History

1 Overview

This document describes the functionality of a Generic File System Driver (FSD) that conforms to the specification required for integration within the Analog Devices' File System Service (FSS). File systems covered by this document include all those that utilize Logical Block Address (LBA) Sector numbers to define locations on the physical media.

A conformant FSD must satisfy the following requirements

1. It must provide for multiple instances of the driver with each instantiation representing a mounted partition, which will be referred to as a *Volume*.
2. Each instance must support the concept of a *current working directory* (CWD) so that requests to either move the CWD or open a file may be expressed relative to the CWD.
3. An internal file descriptor must be created to uniquely identify the pertinent data for each open file. A pointer to the relevant structure must be assigned to the FSD_data_handle member of the `ADI_FSS_FILE_DESCRIPTOR`, FSS File descriptor structure upon opening a file.

   An FSD must return `FALSE` to an `ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT` request.

   An FSD must return either `ADI_FSS_RESULT_SUCCESS` or `ADI_FSS_RESULT_NOT_SUPPORTED` to the `ADI_FSD_CMD_GET_FILE_SYSTEM_SUPPORT` command requesting whether the driver supports the file system as specified in the unique list in the adi_fss.h header file:

```
enum {
      ADI_FSS_FSD_TYPE_UNKNOWN      = 0,
      ADI_FSS_FSD_TYPE_FAT          = 1,
      ADI_FSS_FSD_TYPE_CDDATA_MODE1 = 2,
      ADI_FSS_FSD_TYPE_CDDATA_MODE2 = 3,
      ADI_FSS_FSD_TYPE_CDAUDIO      = 4,
      ADI_FSS_FSD_TYPE_YAFFS        = 5,
};
```

4. An FSD must support all the commands detailed in the "

Compulsory FSD Specific Commands” section (5.4.3).

In the following sections, where an FSD 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. fat

{fsd-ident} = {ns}_{iface} – represents a lower case string, e.g.: `adi_fat`

If any of the above are in uppercase, e.g. {FSD-IDENT} the string is the same as above but in upper case string, e.g.: ADI_FAT

An FSD driver must support both the *block* and *arbitrary* modes of operation, depending on the capabilities of the underlying PID to perform data transfer in the background. Block mode is for use in conjunction with the FSS File Cache when it should be assumed that data read/write operations are requested in blocks, the size of which being dictated by the smallest meaningful unit of file data. In this mode data can be transferred directly between the media and the request buffer, with no intermediate buffer required within the FSD. On the other hand in arbitrary mode the File Cache will not be in operation and the FSD can expect requests for arbitrary amounts of data. (See section 7.2 for further details)

If the PID returns `TRUE` to the `ADI_FSS_CMD_GET_BACKGRND_XFER_SUPPORT`, the FSD must use *block* mode, otherwise the FSD must use *arbitrary* mode. The FSD will be similarly interrogated by the FSS upon opening a file to which it should return the same value as the PID returns.

To use an FSD driver compiled for block mode, the FSS makes a call to adi_dev_Control() to obtain the block size, e.g.:

```
adi_dev_Control( <FSD Device Handle>, ADI_FSD_CMD_GET_BLOCK_SIZE, &BlockSize);
```

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.

<drivers/fsd/{iface}/{fsd-ident}.h This file contains all definitions, function prototypes etc. for the appropriate File System Driver.

<string.h> This file all definitions, function prototypes etc. for the string functions

<ctype.h> This file all definitions, function prototypes etc. for the c type 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/fsd/{iface}/{fsd-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

3.1 Physical Interface Driver (PID)

A peripheral device driver appropriate to the hardware configuration employed is used to transfer data to and from the physical media. The device handle to the PID is passed to the FSD with the following command-value pair, .e.g.:

```
{ ADI_FSD_CMD_SET_PID_HANDLE, (void*)<PID-handle> }
```

The process of issuing the request to the PID must follow the following sequence:

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> },
   ```

3. Add callback function, callback handle and optionally a semaphore[1] to the FSS Super Buffer structure and queue the buffer chain with 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 receipt of the `ADI_PID_EVENT_DEVICE_INTERRUPT` callback for a single buffer (no chain) or upon completion of the last sub-buffer in the chain.

Please refer to the documentation for “Generic PID Design Document” for further details.

[1] The attachment of a semaphore is only required for internal I/O requests. In the case where the FILE Cache is in operation the semaphore is set by the cache module.

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 an FSD, implementers can only specify the heap index for the *cache* heap; an FSD must make use of the general heap defined in the FSS. Thus for all *general* heap usage, the FSD 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 FSD'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.

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

<Fill in the details in the table below. Add further operations if required.>

Operation	**Size (bytes)**
Device Instance. (One instance per partition)	
File Open	
Directory Open	

Table 2 – Dynamic Memory Requirements

4.1 Interrupts

No specific interrupts or interrupt handlers are used by an FSD. The underlying PID may use specific interrupts. Please refer to the documentation of the appropriate PID for further details.

4.2 DMA

This driver does not support DMA directly. The underlying PID may use DMA. Please refer to the documentation of the appropriate PID for further details.

4.3 Timers

No specific timers are used by this driver. Timers and Timer callbacks may be used in the underlying PID. Please refer to the documentation of the appropriate PID for further details.

4.4 Semaphores

The FSD requires two semaphores, one for a Lock Semaphore to maintain exclusive access to the FSD 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

The File System Driver requires the use of the RTC Service.

4.6 Programmable Flags

No Programmable flags are used in this driver. The underlying PID may use programmable flags. Please refer to the documentation of the appropriate PID for further details.

4.7 Pins

No pins are used in this driver. The underlying PID will use pins to communicate with the physical media. Please refer to the documentation for the appropriate PID for further details.

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 3 - Supported Dataflow Directions

5.2 Dataflow Methods

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

ADI_DEV_MODE	Description
ADI_DEV_MODE_CHAINED	Supports the chained buffer method

Table 4 - Supported Dataflow Methods

5.3 Buffer Types

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

- **`ADI_DEV_1D_BUFFER`**
  Linear one-dimensional buffer. This is enveloped by the FSS Super Buffer Structure (Section **Error! Reference source not found.**)
  - `CallbackParameter` – This will always contain the address of the FSS Super Buffer structure.
  - `ProcessedFlag` – This field is not used in the FSD.
  - `pAdditionalInfo` – This field is not used in the FSD.

5.4 Command IDs

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

Commands are sent to the device driver via the `adi_dev_Control()` functionwhich accepts three arguments:

- **DeviceHandle** – This parameter is a `ADI_DEV_DEVICE_HANDLE` type that uniquely identifies the device driver. This handle is provided to the client 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 Commands

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

- **`ADI_DEV_CMD_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)

5.4.3 Compulsory FSD Specific Commands

The command IDs listed below must be supported and processed by the FSD device driver. For read-only file systems the commands relevant to write operations are not required, however.

File Operations:

- **`ADI_FSD_CMD_OPEN_FILE`**
    - The file specified by the `pFullFileName` field of the `ADI_FSS_FILE_DESCRIPTOR` structure specifies the path of the file to be opened. The FSD must allocate memory for its internal file descriptor and add it to its list of open files. On return the FSD must assign the `fsize` field with the size of the file as read from the media, and set the FSD_data_handle field to the address of a pertinent data structure within the FSD that uniquely identifies the open file and its position within the media. If file can not be located and bit 8 of the mode flag in the File Descriptor is set then create file and then treat as if it did exist. If file is located and bit 9 of the mode flag is set then open file and reset to an empty file.
    - Value -The address of the `ADI_FSS_FILE_DESCRIPTOR` structure identifying the file to be opened.

- **`ADI_FSD_CMD_CLOSE_FILE`**
    - Closes the file identified by the `ADI_FSS_FILE_DESCRIPTOR` structure. The FSD must free the memory allocated to its internal *file descriptor* and remove it from its list of open files. On return the FSD must clear the `FSD_data_handle` field of the `ADI_FSS_FSD_FILE_DEF` structure.
    - Value - The address of the `ADI_FSS_FILE_DESCRIPTOR` structure identifying the file to be closed.

- **`ADI_FSD_CMD_SEEK_FILE`**
    - The FSD is to seek to the location in the file as per the values in the `ADI_FSS_SEEK_REQUEST` structure.
    - Value - The address of the `ADI_FSS_SEEK_REQUEST` structure identifying the file to be processed and the seek parameters.

Directory Operations:

- **`ADI_FSD_CMD_CHANGE_DIR`**
    - Adjusts the current working directory location to that specified in the `ADI_FSS_FULL_FNAME` linked-list structure referred to by the associated argument.
    - Value - The address of an `ADI_FSS_FULL_FNAME` structure defining the path name of the directory to which to move.

- **`ADI_FSD_CMD_MAKE_DIR`**
    - Creates a new directory entry in the file system defined by the pathname specified in the `ADI_FSS_FULL_FNAME` linked-list structure referred to by the associated argument. The current working directory in the FSD remains unchanged.

  - Value - The address of an `ADI_FSS_FULL_FNAME` structure defining the path name of the directory to be created.

- **`ADI_FSD_CMD_REMOVE_DIR`**
  - Removes the directory entry in the file system defined by the pathname specified in the `ADI_FSS_FULL_FNAME` linked-list structure referred to by the associated argument. The current working directory in the FSD remains unchanged.
  - Value - The address of an `ADI_FSS_FULL_FNAME` structure defining the path name of the directory to be removed.

- **`ADI_FSD_CMD_OPEN_DIR`**
  - Opens the directory specified by the `pFullFileName` field of the associated `ADI_FSS_FILE_DESCRIPTOR` structure specifies the path of the directory to be opened. The FSD must allocate memory for its internal *file descriptor* and add it to its list of open files. On return the FSD must assign the `fsize` field with the size of the file as read from the media, and set the `FSD_data_handle` field to the address of a pertinent data structure within the FSD that uniquely identifies the open file and its position within the media. Only Directory access commands can be used with a directory so opened; `adi_dev_Read` cannot be used.
  - Value - The address of the `ADI_FSS_DIR_DEF` structure identifying the directory to be opened.

- **`ADI_FSD_CMD_CLOSE_DIR`**
  - Closes the directory identified by the `ADI_FSS_DIR_DEF` structure.
  - Value - The address of the `ADI_FSS_DIR_DEF` structure identifying the directory to be closed.

- **`ADI_FSD_CMD_READ_DIR`**
  - Reads the next directory entry and fills the `struct dirent` structure associated with the `ADI_FSS_DIR_DEF` structure. The `tellpos` field of the `ADI_FSS_DIR_DEF` structure must be set to be the file position of the current entry, and the `curpos` field of the associated `ADI_FSS_FILE_DESCRIPTOR` structure is to point to the location within the directory immediately after the latest directory entry to be read.
  - Value - The address of the `ADI_FSS_DIR_DEF` structure identifying the directory to be read.

- **`ADI_FSD_CMD_SEEK_DIR`**
  - Moves the current position pointer of the open directory to the position specified by `tellpos` field of the `ADI_FSS_DIR_DEF` structure associated argument. On return, the `curpos` field of the associated `ADI_FSS_FILE_DESCRIPTOR` structure is to point to the same location as the by `tellpos` field.
  - Value - The address of the `ADI_FSS_DIR_DEF` structure identifying the directory to be processed.

- **`ADI_FSD_CMD_REWIND_DIR`**

- Rewinds the current position pointer of the open directory to the beginning of the open directory, resetting both the `tellpos` field of the `ADI_FSS_DIR_DEF` structure associated argument and the `curpos` field of the associated `ADI_FSS_FSD_FILE_DEF` structure.
- Value - The address of the `ADI_FSS_DIR_DEF` structure identifying the directory to be rewound.

File System Maintenance Operations:

- **`ADI_FSD_CMD_REMOVE`**
  - Removes the file and associated directory entry in the file system for the file defined by the pathname specified in the `ADI_FSS_FULL_FNAME` linked-list structure referred to by the associated argument. The current working directory in the FSD remains unchanged.
  - Value - The address of an `ADI_FSS_FULL_FNAME` structure defining the path name of the file to be removed.

- **`ADI_FSD_CMD_RENAME`**
  - Renames or relocates the file or directory identified by the `pSource` field of the given `ADI_FSS_RENAME_DEF` structure. The new name or the target directory is identified by the `pTarget` field of the same structure.
  - Value - The address of an `ADI_FSS_RENAME_DEF` structure defining the path names of the file to be renamed and either its new name or the directory to which it is to be relocated.

- **`ADI_FSD_CMD_GET_FILE_SYSTEM_SUPPORT`**
  - Returns ADI_FSS_RESULT_SUCCESS if the FSD supports the required file system as specified by the accompanying 32 bit word; ADI_FSS_RESULT_NOT_SUPPORTED otherwise
  - Value – The unique identifier. Valid enumeration values are supplied in the FSS header file, adi_fss.h.

- **`ADI_FSD_CMD_MOUNT_VOLUME`**
  - Instructs the FSD to read the Boot Record for the volume given by the LBA sector number, and to seek to the root directory.
  - Value - The LBA sector number for the beginning of the required volume, as per the Partition Table.

- **`ADI_FSD_CMD_UNMOUNT_VOLUME`**
  - Instructs the FSD to unmount the volume.
  - Value – N/A.

- **`ADI_FSD_CMD_SET_PID_HANDLE`**
  - Instructs the FSD to use the Device Driver defined by the associated `ADI_DEV_DEVICE_HANDLE` address to read/write data to the physical media.
  - Value - The `ADI_DEV_DEVICE_HANDLE` address identifying the Device Driver to use to read/write data to the physical media.

-

- **`ADI_FSS_CMD_GET_BACKGRND_XFER_SUPPORT`**
    - Requests the FSD to return TRUE or FALSE depending on whether the device supports the transfer of data in the background. The return value will depend on the underlying PID to which this command must be passed on.
    - Value – Client provided location to store result.

- **`ADI_FSS_CMD_GET_DATA_ELEMENT_WIDTH`**
    - Requests the FSD to return the width (in bytes) that defines each data element. The return value will depend on the underlying PID to which this command must be passed on.
    - Value – Client provided location to store result.

- **`ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE`**
    - Requests the FSD 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 FSD 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 FSD instance to use the given Heap Index for any dynamically allocated data caches. Whilst its use is not mandatory, the FSD 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_FSD_CMD_GET_BLOCK_SIZE`**
    - On return the FSD will return the size in bytes of the smallest meaningful unit of data for a file. For many file systems this will be the Cluster size.
    - Value – The address to where the size information is to be stored, on return.

- **`ADI_FSD_CMD_GET_TYPE_STRING`**
    - Instructs the FSD to supply the address of the string describing the driver. This string will be in standard ASCII code.
    - Value – On return, the address of the NULL terminated type string.

- **`ADI_FSD_CMD_GET_LABEL`**
    - Instructs the FSD to supply an address of a text string containing a label of 11 characters. This label will be in standard ASCII code.
    - Value – On return, the address of label string.

5.4.4 Defining FSD Specific Commands

Additional commands may be defined to cater for the specific requirements of the FSD. These commands are only available at configuration time or when the driver is used standalone from the FSS.

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

```
enum {
    {NS}_{IFACE}_CMD_START = ADI_FSD_CUSTOM_CMD_START,   /* (0x000B6000) */
    {NS}_{IFACE}_CMD_<command-description-1>,
    {NS}_{IFACE}_CMD_<command-description-2>,
    {NS}_{IFACE}_CMD_<command-description-n>,
};
```

5.5 Lock Semaphores

An FSD should support at least one Lock Semaphore to prevent multiple threads accessing it at the same time. This Lock Semaphore is granted to a process upon receipt of the following command-value pair,

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

And released upon receipt of the corresponding command-value pair,

```
            { ADI_FSS_CMD_RELEASE_LOCK_SEMAPHORE, NULL },
```

The Lock Semaphore should be created in the `adi_pdd_Open()` function as a binary semaphore with an initial count of one using the Semaphore Service, e.g.:

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

Where `pVolume` is the pointer to the FSD instance data. Thus when a thread acquires a Lock Semaphore the FSD will pend on the *Lock Semaphore* semaphore:

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

and the first thread to do so continues execution. Subsequent threads are thus locked out until the thread holding the Lock Semaphore releases it by posting the *Lock Semaphore* semaphore:

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

The Lock Semaphore must be deleted in the `adi_pdd_Close()` function:

```
      adi_sem_Delete (pVolume->LockSemaphoreHandle);
```

5.6 Semaphores

An FSD should support at least one semaphore to indicate data transfer completion. It should be created in the `adi_pdd_Open()` function as a binary semaphore with an initial count of zero using the Semaphore Service, e.g.:

```
      adi_sem_Create ( 0, &pVolume->DataSemaphoreHandle, NULL );
```

This semaphore handle should be assigned to the `SemaphoreHandle` field of the FSS Super Buffer when data transfer is initiated internally of the FSD. Once the buffer has been queued with the PID the FSD should pend on this semaphore while awaiting transfer completion. Upon receipt of the `ADI_PID_EVENT_DEVICE_INTERRUPT` callback event the FSD should test this value against the one located in the FSS Super Buffer and post it on a match. See Section 5.7 for more details.

5.7 Callback Events

An FSD driver does not generate callback events, as it simply processes buffers supplied by the FSS and passes them on to a PID, which will generate a callback event of data completion. However an FSD must supply a callback function to be called from the FSS in response to the `ADI_DEV_EVENT_BUFFER_PROCESSED` and `ADI_PID_EVENT_DEVICE_INTERRUPT` events. In addition it must supply a meaningful handle to be passed as the first argument in the callback function. Typically this handle will be the address of the FSD instance data. These values must be assigned to the `FSDCallbackFunction` and `FSDCallbackHandle` fields in the FSS Super buffer before queuing the buffer chain with the PID.

The other arguments to the callback function are as for all functions of type ADI_DCB_CALLBACK_FN:

```
void FSDCallback( void* Handle, u32 Event, void *pArg );
```

The FSD can assume that the pArg value points to the FSS Super Buffer structure of the sub buffer for which data transfer has completed.

If the FSD owns the semaphore located in the FSS Super buffer then it must post it in response to the `ADI_PID_EVENT_DEVICE_INTERRUPT` event. Then in response to the same event it must either submit the next LBA request in the chain (it the latter's `SectorCount` value is not zero) or release the PID Lock Semaphore if at the end of the chain. For standalone use, deadlock occurs if the semaphore is posted in a separate callback to that which releases the PID Lock Semaphore[2].

5.8 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
      }
```

5.8.1 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`**

[2] This may vary with other RTOSes but should be safe to maintain the same control in each case.

The function executed successfully.

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

5.8.2 Device Driver Specific Return Codes

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

- **`ADI_FSS_RESULT_BAD_NAME`**
  The file/directory name specified is invalid.
- **`ADI_FSS_RESULT_NOT_FOUND`**
  The specified file/directory cannot be located in the file system.
- **`ADI_FSS_RESULT_OPEN_FAILED`**
  The file specified cannot be opened, due to an error condition.
- **`ADI_FSS_RESULT_CLOSE_FAILED`**
  The file specified cannot be closed.
- **`ADI_FSS_RESULT_MEDIA_FULL`**
  The operation cannot be completed because the physical media is full.
- **`ADI_FSS_RESULT_NO_MEMORY`**
  There is insufficient memory to satisfy a dynamic allocation request.

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 FSD source module, {fsd-ident}.c, and declared as an extern variable in the FSD header file, {fsd-ident}.h, where its presence is guarded from inclusion in the FSD source module as follows:

- In the source module and ahead of the #include statement for the header file define the macro, `__{FSD-IDENT}_C__`.
- In the header file, guard the extern declaration:

```
#if !defined(__{FSD-IDENT}_C__)
extern ADI_DEV_PDD_ENTRY_POINT {FSD_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 FSD header file, {fsd-ident}.h, and guarded against inclusion in the FSD Source module, and should only only be to included in an application module if the developer defines the macro, `_{FSD-IDENT}_DEFAULT_DEF_`:

```
#if !defined(__{FSD-IDENT}_C__)
:
#if defined(_{FSD-IDENT}_DEFAULT_DEF_)
static ADI_DEV_CMD_VALUE_PAIR {FSD-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 all FSDs 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 an FSD its value should be assigned to `&{FSD_IDENT}_EntryPoint`.
`pConfigTable`	This is a pointer to the table of command-value pairs to configure the FSD, and its value should be assigned to `{FSD_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 FSDs 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 FSD.

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

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

The following data structures are used in conjunction with the above commands. All but one of these are defined in the `<services/fss/adi_fss.h>` header file.

6.4 FSS File Descriptor, ADI_FSS_FILE_DESCRIPTOR

This structure is passed to an FSD for all operations on open files. It is defined in the FSS header file, adi_fss.h as:

```
typedef struct {
    ADI_FSS_FULL_FNAME        *pFullFileName;
    u32                       curpos;
    u32                       fsize;
    int                       mode;
    ADI_FSS_FSD_DATA_HANDLE   FSD_data_handle;
    ADI_DEV_DEVICE_HANDLE     FSD_device_handle;
    void                      *pCriticalRegionData;
    ADI_FSS_CACHE_DATA_HANDLE Cache_data_handle;
} ADI_FSS_FILE_DESCRIPTOR;
```

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

`pFullFileName`	Linked list containing the full path name.
`curpos`	Current byte position within the open file.
`fsize`	The total file size in bytes. On file-open this value must be set to the value recorded in the files' directory entry.
`mode`	The mode for which the file is opened. Section details the appropriate modes.
`FSD_data_handle`	The FSD must assign the address of an internal data structure that uniquely identifies the status of the open file in the appropriate terms of the File System.
`FSD_device_handle`	This must be the Device Handle identifying the FSD device driver and must be the same as the third argument in the call to adi_pdd_Open().
`pCriticalRegionData`	Critical region data pointer. Currently not used.
`Cache_data_handle`	This handle is reserved for use with the File Cache module.

6.5 ADI_FSS_FULL_FNAME

Contains a linked list defining the path name of a file. If the path is absolute then the name field of the first entry in the linked list will be NULL, otherwise the path is to be interpreted as being relative to the current working directory. It is defined in the adi_fss.h header file as:

```
typedef struct ADI_FSS_FULL_FNAME {
    struct ADI_FSS_FULL_FNAME   *pNext;
    struct ADI_FSS_FULL_FNAME   *pPrevious;
    ADI_FSS_WCHAR               *name;
    u32                         namelen;
} ADI_FSS_FULL_FNAME;
```

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

`pNext`	The next item in the linked list
`pPrevious`	The previous item in the linked list
`name`	The name of the current path element (directory or file name)
`namelen`	The length of the current path element

6.6 The FSS Super Buffer Structure, ADI_FSS_SUPER_BUFFER

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

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

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

The fields in black are set by the originating module, which also must zero the fields associated with the modules that are not used in the submission of the buffer. For example, the internal FSD request to read the a directory block from zero the red and blue fields,.

The definition of the structure is:

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

Where the fields are defined as:

`Buffer`	The `ADI_DEV_1D_BUFFER` structure required for the transfer. Please note that this is not a pointer field. This should only be set by the FSD if it is originating the data transfer request.
`SemaphoreHandle`	The Handle of the Semaphore to be posted upon completion of data transfer. This should only be set by the FSD if it is originating the data transfer request, when it should be set to the value stored in the FSD instance data. See section below for use of semaphores.
`LBARequest`	The `ADI_FSS_LBA_REQUEST` structure for the associated buffer. The FSD is responsible for setting the values for this structure, whether the request is internally generated or passed from the File Cache module. If the buffer forms part of a chain and it can be shown that several sub buffers are contiguous on the media the FSD can optionally combine the LBA requests to cover a number of sub buffers. In which case the `SectorCount` value of each sub buffer that is represented by an LBA request of a previous sub buffer must be set to zero.
`pBlock`	Used in the File Cache. Its value must remain unchanged by the FSD. For internal FSD transfers it must to set to NULL.
`LastinProcessFlag`	Used in the File Cache. Its value must remain unchanged by the FSD. For internal FSD transfers it must to set to NULL.
`pFileDesc`	Used in the File Cache. Its value must remain unchanged by the FSD. For internal FSD transfers it must to set to NULL.
`FSDCallbackFunction`	The FSD must assign the address of the callback function to be invoked on the transfer completion events.
`FSDCallbackHandle`	The FSD must assign the address of a pertinent structure to be passed as the first argument in the call to the function defined by the `FSDCallbackFunction` field.
`PIDCallbackFunction`	This handle is reserved for use with PIDs.
`PIDCallbackHandle`	This handle is reserved for use with PIDs.

6.7 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 information is made available to the FSD upon mounting.
`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.

6.8 ADI_FSS_VOLUME_DEF

This structure contains the information required to mount the appropriate File System on the volume defined. 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`	File System Type of volume. Should agree with an identifier stored in the FSS module. See section for details on supported File System types.
`StartAddress`	Start address of volume on media, in LBA Sector format.
`VolumeSize`	Number of Sectors in volume.
`SectorSize`	Number of bytes per sector in volume.
`DeviceNumber`	The number of the device on the bus. This value must be used for the `DeviceNumber` field in the LBA request structure.

6.9 ADI_FSS_SEEK_REQUEST

Contains the seek parameters and the file within which to seek. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
    ADI_FSS_FILE_DESCRIPTOR *pFileDesc;
    int whence;
    long offset;
} ADI_FSS_SEEK_REQUEST;
```

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

`pFileDesc`	Pointer to the FSS File Descriptor of the file to be manipulated.
`whence`	Flag determining the start point for the seek operation: 0 – Seek form start of file, 1 – seek relative to current position, 2 – seek from end of file.
`Offset`	The number of bytes from the seek start point.

6.10 ADI_FSS_DIR_DEF

Contains the information relevant to an open directory. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
    ADI_FSS_FILE_DESCRIPTOR *pFileDesc;
    ADI_FSS_DIR_ENTRY entry;
    u32 tellpos;
} ADI_FSS_DIR_DEF;
```

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

`pFileDesc`	Pointer to the FSS File Descriptor of the directory to be manipulated.
`entry`	The details of the current entry. The ADI_FSS_DIR_ENTRY is simply a typedef of the struct dirent entry defined in the dirent.h header file and detailed in section 6.11. The FSD must populated this structure with data interpreted from the associated file system specific directory entry.
`tellpos`	The position within the file of the current directory entry.

6.11 ADI_FSS_DIR_ENTRY

Contains the information relevant to the current directory entry. It is defined in the FSS header file, adi_fss.h, as:

```
struct dirent {
      ino_t d_ino;
      off_t d_off;
      unsigned char d_namlen;
      unsigned char d_type;
      char d_name[256];
      u32 d_size;
      struct tm DateCreated;
      struct tm DateModified;
      struct tm DateLastAccess;
};
```

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

`d_ino`	File Serial Number
`d_off`	Offset to next directory entry
`d_namlen`	length minus trailing \0 of entry name
`d_type`	Type – DT_REG for a regular file, or DT_DIR for a sub directory,
`d_name`	Entry name, 256 characters maximum.
`d_size`	File Size in bytes.
`DateCreated`	Date & Time when entry was created.
`DateModified`	Date & Time when entry was last modified.
`DateLastAccess`	Date & Time when entry was last accessed.

6.12 ADI_FSS_RENAME_DEF

Defines the source and target names for a rename operation. If the source is a file and the target is a directory then the source file will simply be moved to the target directory. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
    ADI_FSS_FULL_FNAME      *pSource;
    ADI_FSS_FULL_FNAME      *pTarget;
} ADI_FSS_RENAME_DEF;
```

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

`pSource`	Pointer to the linked list containing the path of the source file/directory.
`pTarget`	Pointer to the linked list containing the path of the target file/directory.

6.13 File System Types.

The following enumeration gives the unique values to be used by an FSD to identify the file system it supports:

```
enum {
      ADI_FSS_FSD_TYPE_UNKNOWN      = 0,
      ADI_FSS_FSD_TYPE_FAT          = 1,
      ADI_FSS_FSD_TYPE_CDDATA_MODE1 = 2,
      ADI_FSS_FSD_TYPE_CDDATA_MODE2 = 3,
      ADI_FSS_FSD_TYPE_CDAUDIO      = 4,
      ADI_FSS_FSD_TYPE_UDF          = 5,
      ADI_FSS_FSD_TYPE_YAFFS        = 6,
};
```

Where the file systems are:

`ADI_FSS_FSD_TYPE_UNKNOWN`	Unknown file system
`ADI_FSS_FSD_TYPE_FAT`	FAT 12/16/32.
`ADI_FSS_FSD_TYPE_CDDATA_MODE1`	ISO 9660 compact disk Yellow Book Data format for Mode 1 and Mode 2 Form 1.
`ADI_FSS_FSD_TYPE_CDDATA_MODE2`	ISO 9660 compact disk Yellow Book Data format for Mode Mode 2 Form 2.
`ADI_FSS_FSD_TYPE_CDAUDIO`	ISO 9660 compact disk Red Book Data format for CD Audio data.
`ADI_FSS_FSD_TYPE_YAFFS`	Yet another Flash File System by Aleph One for NAND flash.

7 Opening and Configuring the File System Driver

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

7.1 Procedure for Opening

The File System Service (FSS) will automatically open the appropriate FSD by issuing a call to `adi_dev_Open()` upon detecting the presence of a data volume. The arguments to this call are supplied by the `ADI_FSS_DEVICE_DEF` structure (section 6.3).

Next the FSD will be sent the `ADI_FSD_CMD_GET_FILE_SYSTEM_SUPPORT` command with one of the values defined in section 7.7. The FSD should compare this with its internal value and return `ADI_FSS_RESULT_SUCCESS` upon a match.

If unsuccessful the FSD will be closed, otherwise the remaining commands are received in the following order:

1. `ADI_DEV_CMD_SET_DATAFLOW_METHOD` – The dataflow method is set to `ADI_DEV_MODE_CHAINED` as mandatory for Device Drivers.
2. `ADI_DEV_CMD_TABLE` – here the address of the configuration table defined by the user and assigned to the `pConfigTable` field of the `ADI_FSS_DEVICE_DEF` structure (section 6.3) is passed to the FSD for configuration.
3. `ADI_FSD_CMD_SET_PID_HANDLE` – the Device Handle of the lower level PID is passed to the FSD to enable it to make calls on the PID device driver. The FSD should determine at this point the Data Element Width supported by the PID and whether the PID supports the background transfer of data.
4. `ADI_FSD_CMD_MOUNT_VOLUME` – Finally the address of an ADI_VOLUME_DEF structure is passed to the FSD with all the information required to mount the appropriate file system on the media.

7.2 Defining the Configuration Structures

The FSD must define defaults for both the `ADI_FSS_DEVICE_DEF` structure and the `ADI_DEV_CMD_VALUE_PAIR` configuration table in the {fsd-ident}.h header file. These should be guarded with the `_{FSD-IDENT}_DEFAULT_DEF_` macro such that if the macro is not defined then neither are the definitions. In this way doing nothing means that the user of the FSD must specify the structures themselves, and there will be no duplicate symbols to worry about. For example

```
#if defined(_{FSD-IDENT}_DEFAULT_DEF_)

static ADI_FSS_DEVICE_DEF {FSD_IDENT}_Def = {
    0,
    &ADI_FAT_EntryPoint,
    NULL,
    NULL,
    ADI_DEV_DIRECTION_BIDIRECTIONAL,
    NULL
};
#endif
```

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

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

7.3 Default Settings

There are no default settings for this type of device driver.

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
File System	Unique File System identifier.	ADI_FSD_CMD_GET_FILE_SYSTEM_SUPPORT
Set PID Handle	Device Handle of Physical Interface Driver.	ADI_FSD_CMD_SET_PID_HANDLE
Mount Volume	Sector Number at start of the required partition	ADI_FSD_CMD_MOUNT_VOLUME
Get Block Size	Address of u32 variable to store result.	ADI_FSD_CMD_GET_BLOCK_SIZE
Get Callback function	Address of location to store the address of the callback function.	ADI_FSD_CMD_GET_CALLBACK_FUNCTION

Table 5 – Additional Required Settings

8 Hardware Considerations

There are no hardware considerations for a file system driver. However, the underlying Physical interface driver will have particular hardware requirements. Please refer to the documentation for the appropriate PID for further details.