ADI FAT FILE SYSTEM DRIVER

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
22 August 2006	Added dynamic memory requirements, and changed command syntax, added block mode vs arbitary mode discussion, etc.
1 November 2006	Amended control commands and updated configuration details.
18 July 2007	Updated to final release specification.
20 July 2007	Replaced ‘mutex’ with ‘lock semaphore’ throughout

Table 1 - Revision History

1 Overview

This document describes the functionality of the ADI FAT File System Driver (FSD) that conforms to the specification required for integration within the Analog Devices' File System Service (FSS). The FAT file system described in this document is compatible with volumes formatted with either 12, 16 or 32 bit FAT file systems and which utilize Logical Block Address (LBA) Sector numbers to define locations on the physical media.

The FAT FSD satisfies the following requirements:

1. It provides for multiple instances of the driver with each instantiation representing a mounted partition, which will be referred to as a *Volume*.
2. Each FAT driver instance supports 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 is defined to uniquely identify the pertinent data for each open file. A pointer to this internal structure is assigned to the `FSD_data_handle` member of the `ADI_FSS_FILE_DESCRIPTOR` structure upon opening a file.
4. The FAT FSD returns `FALSE` to an `ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT` request.
5. The FAT FSD returns either `ADI_FSS_RESULT_SUCCESS` to the

   `ADI_FSD_CMD_GET_FILE_SYSTEM_SUPPORT, (void*)FileSystemType }`

   Command when the accompanying value, `FileSystemType,` is `ADI_FSS_FSD_TYPE_FAT(1)` and `ADI_FSS_RESULT_NOT_SUPPORTED` for all other values.
6. The FAT FSD supports all the commands detailed in the Section 6.4.3.
7. The FAT FSD driver dynamically supports both *block* and *arbitrary* modes of operation, depending on the capabilities of the underlying PID to perform data transfer in the background, as determined by the response to the `ADI_FSS_CMD_GET_BACKGRND_XFER_SUPPORT` command. The FAT driver passes this response on the FSS when it receives the `ADI_FSS_CMD_GET_BACKGRND_XFER_SUPPORT` command itself.

   Block mode is for use in conjunction with the FSS File Cache when it should be assumed that data transfer operations are requested in blocks, the size of which being dictated by the cluster size of the mounted partition. In this mode data can be transferred directly between the media and the request buffer, with no intermediate buffer required within the FAT FSD. The FAT FSD supplies the cluster size to the FSS in response to the `ADI_FSD_CMD_GET_BLOCK_SIZE` command.

   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. In this case the FAT FSD maintains a buffer to hold the contents of the current file cluster.

2 Quick Start Guide

2.1 Reference Chart for System Services Initialization:

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

Secondary Interrupt Handlers	None.
DMA channels	None.
Device Drivers	One per FAT volume.
Semaphores	Two per FAT volume.
Flag Callbacks	None.

2.2 Registering the FAT FSD with the File System Service

To add the FAT FSD to the FSS, include the FAT FSD header file, adi_fat.h in the application source code, optionally choosing to accept the default definition structure, `ADI_FAT_Def`, e.g:

```
#define _ADI_FAT_DEFAULT_DEF_
#include <drivers/fsd/fat/adi_fat.h>
```

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

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

For details on providing a custom definition please refer to Section 8.1.

2.3 Dynamic Memory Requirements (FAT16/32)

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

Operation	**Size (bytes)**
FAT Driver Instantiation	
Device Instance. (One instance per partition) - base	248
+ mounted FAT16/32	+ 512 + cluster size
Temporary BPB buffer (during mount operation only)	+ 512
File Access	
File Open – FAT 16/32 – base requirement / block mode	596
File Open – FAT 16/32 – additional for arbitrary mode	cluster size
File Open – FAT 16/32 – additional for write mode	512 + cluster size
Temporary Directory buffers (for directory search)	512 + cluster size
Directory Access	
Directory Open - FAT 16/32	596 + cluster size
Temporary Directory buffers (for directory search)	512 + cluster size

3 Files

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

3.1 Include Files

The FAT driver source includes the following header 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/fat/adi_fat.h**
  This file contains all definitions, function prototypes etc. for the appropriate Physical Interface Driver.

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

- **<ctype.h>**
  This file all definitions, function prototypes etc. for the c type functions.

3.2 Source Files

The FAT driver source code is contained in a single file, Blackfin/lib/src/drivers/fsd/adi_fat.c. This file can be optionally compiled with the `ADI_USE_FAT_FORMAT` macro defined to expose the formatting functions. It is advised, however, that this macro is left undefined for inclusion in the distributed device driver libraries.

4 Lower Level Drivers

4.1 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 FAT FSD with the following command-value pair, .e.g.:

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

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

5 Resources Required

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

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

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

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

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

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

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

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

Operation	**Size (bytes)**
FAT Driver	
Device Instance. (One instance per partition) - base	248
+ mounted FAT16/32	+ 512 + cluster size
+ mounted FAT12	+ 1024 + cluster size
Temporary BPB buffer (during mount operation only)	+ 512
File Access	
File Open – FAT12 – base requirement / block mode	1108

File Open – FAT 16/32 – base requirement / block mode	596
File Open – FAT12 – additional for arbitrary mode	1084 + cluster size
File Open – FAT 16/32 – additional for arbitrary mode	512 + cluster size
File Open – FAT 12 – additional for write mode	1084 + cluster size
File Open – FAT 16/32 – additional for write mode	512 + cluster size
Temporary Directory buffers (for directory search) – FAT12	1084 + cluster size
Temporary Directory buffers (for directory search) – FAT16/32	512 + cluster size
Directory Access	
Directory Open - FAT 12	1108 + cluster size
Directory Open - FAT 16/32	596 + cluster size
Temporary Directory buffers (for directory search)	512 + cluster size

Table 2 – Dynamic Memory Requirements

The following table gives an example for a FAT16 volume, with a Cluster size of 8192 bytes (257-512MB volume):

	Peak	**Sustained**
Mounted volume	9464	8952
Open File (read) – Block mode	9300	596
Open File (read) – Arbitrary mode	17492	8788

5.1 Interrupts

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

5.2 DMA

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

5.3 Timers

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

5.4 Semaphores

Each instance of the FAT 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 is used to create and manipulate all semaphores.

5.5 Real-Time Clock

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

5.6 Programmable Flags

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

5.7 Pins

No pins are used in the FAT 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.

6 Supported Features of the Device Driver

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

6.1 Directionality

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

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

Table 3 - Supported Dataflow Directions

6.2 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

6.3 Buffer Types

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

- **ADI_DEV_1D_BUFFER**
  Linear one-dimensional buffer. This is enveloped by the FSS Super Buffer.
  - `CallbackParameter` – This will always contain the address of the FSS Super Buffer structure.
  - `ProcessedFlag` - This field is not used in the FAT FSD.
  - `pAdditionalInfo` – This field is not used in the FAT FSD.

6.4 Command IDs

This section enumerates the commands that are supported by the FAT 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 FAT driver supports. The remaining section describes the mandatory FSD specific commands employed by the FAT driver.

Commands are sent to the device driver via the `adi_dev_Control()` function which 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.

6.4.1 Device Manager Commands

The commands listed below are supported and processed directly by the Device Manager. As such, all device drivers support these commands.

- **ADI_DEV_CMD_TABLE**
  - Table of command pairs being passed to the driver
  - Value – ADI_DEV_CMD_VALUE_PAIR *
- **ADI_DEV_CMD_END**
  - Signifies the end of a command pair table
  - Value – ignored
- **ADI_DEV_CMD_PAIR**
  - Single command pair being passed
  - Value – ADI_DEV_CMD_PAIR *
- **ADI_DEV_CMD_SET_SYNCHRONOUS**
  - Enables/disables synchronous mode for the driver
  - Value – TRUE/FALSE
- **ADI_DEV_CMD_GET_INBOUND_DMA_CHANNEL_ID**
  - Returns the DMA channel ID value for the device driver's inbound DMA channel
  - Value – u32 * (location where the channel ID is stored)

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

6.4.2 Common 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 – address of location where the result is to be stored. The FAT driver returns FALSE to this command.

6.4.3 Mandatory FSD Specific Commands

The command IDs listed below are supported and processed by the FAT 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 FAT FSD allocates memory for its internal file descriptor and assigns its address to the `FSD_data_handle` field of the `ADI_FSS_FILE_DESCRIPTOR` structure. On return the FSD assigns the `fsize` field with the size of the file as read from the media. If the file cannot be located and bit 8 of the mode flag in the FSS File Descriptor is set then the file will be created. If the file is located and bit 9 of the mode flag is set then the file is opened and truncated 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 FAT FSD frees the memory allocated to its internal *file descriptor* and data buffers and clears 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 FAT FSD is requested 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 FAT FSD allocates memory for its internal file descriptor and assigns its address to the FSD_data_handle field of the ADI_FSS_FILE_DESCRIPTOR structure. On return the FSD assigns the fsize field with the size of the file as read from 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 accompanying 32 bit word has the value `ADI_FSS_FSD_TYPE_FAT` (`1`) as per the enumeration values are supplied in the FSS header file, adi_fss.h. Returns `ADI_FSS_RESULT_NOT_SUPPORTED` otherwise.
  - Value – The unique identifier.

- **ADI_FSD_CMD_MOUNT_VOLUME**
  - Instructs the FAT FSD to read the BIOS Parameter Block for the volume given by the LBA sector number, and to position the driver at 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 FAT FSD to unmount the volume.
  - Value – N/A.

- **ADI_FSD_CMD_SET_PID_HANDLE**
  - Instructs the FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT FSD instance to use the given Heap Index for any dynamically allocated data caches. The default behavior is to use the FSS General Heap.
  - Value – the Index of the required heap.
- **ADI_FSD_CMD_GET_BLOCK_SIZE**
  - On return the FAT FSD will return the Cluster size in bytes.
  - Value – The address to where the size information is to be stored, on return.
- **ADI_FSD_CMD_GET_TYPE_STRING**
  - Instructs the FAT FSD to supply the address of the string describing the driver.
  - Value – On return, the address of the NULL terminated type string, "FAT12", "FAT16", or "FAT32".
- **ADI_FSD_CMD_GET_LABEL**
  - Instructs the FAT 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, which will be label identified by either the volume ID entry in the root directory or that identified in the BIOS parameter block of the mounted volume.

6.4.4 Defining FAT FSD Specific Commands

No additional commands are defined for the FAT FSD.

6.5 Lock Semaphore

The FAT FSD supports 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 },
```

6.6 Semaphores

The FAT FSD supports one semaphore to indicate data transfer completion. This semaphore handle is assigned to the `SemaphoreHandle` field of the FSS Super Buffer when data transfer is initiated internally of the FAT FSD. Once the buffer has been queued with the PID the FSD pendson this semaphore while awaiting transfer completion. Upon receipt of the `ADI_PID_EVENT_DEVICE_INTERRUPT` callback event the FAT FSD will test this value against the one located in the FSS Super Buffer and post it on a match. See Section 6.7 for more details.

6.7 Callback Events

The FAT 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 upon completion of data transfer. However the FAT FSD supplies 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 generated by the

PID. In addition it supplies 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 are 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 FAT FSD assumes that the pArg value points to the FSS Super Buffer structure of the sub buffer for which data transfer has completed.

If the FAT FSD owns the semaphore located in the FSS Super buffer then it will post it in response to the `ADI_PID_EVENT_DEVICE_INTERRUPT` event. Then in response to the same event it will 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.

6.8 Return Codes

All API functions of the FAT FSD 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 non-zero values.

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

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

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

6.8.1 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`**
  The function executed successfully.
- **`ADI_DEV_RESULT_NOT_SUPPORTED`**
  The function is not supported by the driver.
- **`ADI_DEV_RESULT_DEVICE_IN_USE`**
  The requested device is already in use.
- **`ADI_DEV_RESULT_NO_MEMORY`**
  There is insufficient memory available.

- **ADI_DEV_RESULT_BAD_DEVICE_NUMBER**
  The device number is invalid.
- **ADI_DEV_RESULT_DIRECTION_NOT_SUPPORTED**
  The device cannot be opened in the direction specified.
- **ADI_DEV_RESULT_BAD_DEVICE_HANDLE**
  The handle to the device driver is invalid.
- **ADI_DEV_RESULT_BAD_MANAGER_HANDLE**
  The handle to the Device Manager is invalid.
- **ADI_DEV_RESULT_BAD_PDD_HANDLE**
  The handle to the physical driver is invalid.
- **ADI_DEV_RESULT_INVALID_SEQUENCE**
  The action requested is not within a valid sequence.
- **ADI_DEV_RESULT_ATTEMPTED_READ_ON_OUTBOUND_DEVICE**
  The client attempted to provide an inbound buffer for a device opened for outbound traffic only.
- **ADI_DEV_RESULT_ATTEMPTED_WRITE_ON_INBOUND_DEVICE**
  The client attempted to provide an outbound buffer for a device opened for inbound traffic only.
- **ADI_DEV_RESULT_DATAFLOW_UNDEFINED**
  The dataflow method has not yet been declared.
- **ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE**
  The dataflow method is incompatible with the action requested.
- **ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE**
  The device does not support the buffer type provided.
- **ADI_DEV_RESULT_NON_TERMINATED_LIST**
  The chain of buffers provided is not NULL terminated.
- **ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED**
  No callback function was supplied when it was required.
- **ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE**
  Requires the device be opened for bidirectional traffic only.

6.8.2 Device Driver Specific Return Codes

The return codes listed below are supported and processed by the FAT 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.

7 Data Structures

7.1 Device Driver Entry Points, ADI_DEV_PDD_ENTRY_POINT

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

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

```
#if !defined(__ADI_FAT_C__)
extern ADI_DEV_PDD_ENTRY_POINT ADI_FAT_EntryPoint;
:
#endif
```

7.2 Command-Value Pairs, ADI_DEV_CMD_VALUE_PAIR

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

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

No default table is declared in the FAT FSD header file, adi_fat.h.

7.3 Device Definition Structure, ADI_FSS_DEVICE_DEF

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

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

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

DeviceNumber	This defines which peripheral device to use. This is the DeviceNumber argument required for a call to adi_dev_Open(). This value is ignored by the FAT FSD.
pEntryPoint	This is a pointer to the device driver entry points and is passed as the pEntryPoint argument required for a call to adi_dev_Open(). For the FAT FSD its value should be assigned to &ADI_FAT_EntryPoint.
pConfigTable	This is a pointer to the table of command-value pairs to configure the FAT FSD; the default value for the FAT FSD is NULL.
pCriticalRegionData	This is a pointer to the argument that should be passed to the System Services adi_int_EnterCriticalRegion() function. This is currently not used and should be set to NULL.
Direction	This is the Direction argument required for a call to adi_dev_Open(). For the FAT FSDs this value should 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 the FAT FSD.

A default instantiation of this structure is declared in the FSD header file, adi_fat.h, and guarded against inclusion in the FSD Source module, and will only be included in an application module if the developer defines the macro, _ADI_FAT_DEFAULT_DEF_:

```
#if !defined(__ADI_FAT_C__)
:
#if defined(_ADI_FAT_DEFAULT_DEF_)
static ADI_FSS_DEVICE_DEF ADI_FAT_Def = { … };
:
#endif
:
#endif
```

7.4 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 FAT FSD assigns the address of its internal data structure that uniquely identifies the status of the open file in appropriate terms.
FSD_device_handle	This must be the Device Handle identifying the FAT 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.

7.5 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

7.6 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_1D_BUFFER` structure as its first member, the two structures share addresses, such that

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

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

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

The definition of the structure is:

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

Where the fields are defined as:

`Buffer`	The `ADI_DEV_1D_BUFFER` structure required for the transfer. Please note that this is not a pointer field. This is only set by the FAT FSD if it is originating the data transfer request.
`SemaphoreHandle`	The Handle of the Semaphore to be posted upon completion of data transfer. This is only set by the FAT FSD if it is originating the data transfer request, when it is set to the value stored in the FAT FSD instance data. See section below for use of semaphores.
`LBARequest`	The `ADI_FSS_LBA_REQUEST` structure for the associated buffer. The FAT 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 FAT FSD will 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 remains unchanged by the FAT FSD. For internal FAT FSD transfers it is set to NULL.
LastinProcessFlag	Used in the File Cache. Its value remains unchanged by the FAT FSD. For internal FSD transfers it is set to NULL.
pFileDesc	Used in the File Cache. Its value must remain unchanged by the FSD. For internal FAT FSD transfers it is set to NULL.
FSDCallbackFunction	The FAT FSD assigns the address of the callback function to be invoked on the transfer completion events.
FSDCallbackHandle	The FAT FSD assigns 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.

7.7 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 FAT FSD, via the ADI_FSS_VOLUME_DEF structure, 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.

7.8 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.

7.9 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.

7.10 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 7.11. The FAT FSD will populate this structure with data interpreted from the associated file system specific directory entry.
tellpos	The position within the file of the current directory entry.

7.11 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.

7.12 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.

7.13 ADI_FSS_FORMAT_DEF

Defines details required to format a given partition. It is defined in the FSS header file, adi_fss.h, as:

```
typedef struct {
      ADI_FSS_VOLUME_IDENT ident;
      ADI_FSS_WCHAR        *label;
      u32                  label_len;
      u32                  OptionMask;
      ADI_FSS_VOLUME_DEF   ;
} ADI_FSS_FORMAT_DEF;
```

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

ident	The unique identifier for the mounted volume. This field is ignored by the FAT FSD.
label	The label that the partition is to take, up to 11 asci characters. This label is written as the first directory (attribute value = 0x08) entry in the root directory. It never appears in directory listings.
label_len	The length of the label.
OptionMask	This field defines the file system type to be implemented. The value is defined by the `ADI_FSS_FMT_OPTION_VALUE(FILESYS,TYPE)` macro defined in adi_fss.h, where `FILESYS` must always be `ADI_FSS_FSD_TYPE_FAT` (see Section 0) for the FAT FSD and the `TYPE` value must be `0` for FAT12, `1` for FAT16 and `2` for FAT32.
VolumeDef	The `ADI_FSS_VOLUME_DEF` structure defining the partition. See Section 0 for details.

7.14 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.

8 Opening and Configuring the FAT File System Driver

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

8.1 Registering the FAT FSD with the File System Service

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

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

This structure will require the address of the FAT FSD entry point structure, `ADI_FAT_EntryPoint` (Section 7.1), which is define in the FAT FSD header file, <drivers/fsd/fat/adi_fat.h>. An example configuration table (Section 7.2) and definition structure could be:

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

```
ADI_FSS_DEVICE_DEF ADI_FAT_Def = {
            0,
            &ADI_FAT_EntryPoint,
            ADI_FAT_ConfigTable,
            NULL,
            ADI_DEV_DIRECTION_BIDIRECTIONAL,
            'C'
};
```

Alternatively, the default definition structure, defined in the FAT FSD header file, can be used by defining the `_ADI_FAT_DEFAULT_DEF_` macro ahead of including the header file:

```
#define _ADI_FAT_DEFAULT_DEF_
#include <drivers/fsd/fat/adi_fat.h>
```

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

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

8.2 Procedure for Opening

The File System Service (FSS) will automatically open the FAT FSD by issuing a call to `adi_dev_Open()` upon detecting the presence of data volume. The arguments to this call are supplied by the `ADI_FSS_DEVICE_DEF` structure (section 7.3).

Next the FAT FSD will be sent the `ADI_FSD_CMD_GET_FILE_SYSTEM_SUPPORT` command with one of the values defined in section 7.7. The FAT FSD should compare this with its internal value and return `ADI_FSS_RESULT_SUCCESS` if the value equals the `ADI_FSS_FSD_TYPE_FAT` enumeration value.

If unsuccessful the FAT 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 7.3) is passed to the FAT FSD for configuration.
3. `ADI_FSD_CMD_SET_PID_HANDLE` – the Device Handle of the lower level PID is passed to the FAT FSD to enable it to make calls on the PID device driver.
4. `ADI_FSD_CMD_MOUNT_VOLUME` – Finally the address of an `ADI_VOLUME_DEF` structure is passed to the FAT FSD with all the information required to mount the FAT file system on the media.

8.3 Default Settings

The following default values are assumed:

Item	Default Value	Command ID
Cache Heap ID	-1	`ADI_FSS_CMD_SET_CACHE_HEAP_ID`

8.4 Additional Required Configuration Settings

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

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

Table 5 – Additional Required Settings

9 Hardware Considerations

There are no hardware considerations for a FAT FSD. However, the underlying Physical Interface Driver will have particular hardware requirements. Please refer to the documentation for the appropriate PID for further details.