**ANALOG
DEVICES**

ADI RAM DISK
PHYSICAL INTERFACE DRIVER

DATE: 6 DECEMBER 2010

Table of Contents

List of Tables

Acronyms

ADI   Analog Devices Inc.

ADSP – BF   Analog Devices Digital Signal Processor – Blackfin

DMA   Direct Memory Access

Document Revision History

Date	Description of Changes
01-Sep -10	Initial release

Table 1 – Revision History

1. Overview

This document describes the functionality of the RAM Disk physical interface driver (PID). The driver adheres to Analog Devices' File System Service (FSS) Model and provides access to RAM Disk devices. Out of the box support is available for all Blackfin processors for which the Services and Drivers are supported. The RAM Disk PID can also be used as a configurable alternative to the built-in RAM Disk provided with the USB mass storage class driver.

2. Quick Start Guide

This section is provided to get you up and running with the RAM Disk PID driver as quickly as possible using the default configuration options.

2.1. Reference Chart for System Services Initialization:

The following tables detail the System Services & Device Driver resources as required to be set in the adi_ssl_init.h header file. These include the requirements for the lower level drivers.

DMA channels	0
Device Drivers	1
Semaphores	2
Flag Callbacks	None
Interrupts	
None	

2.2. Assigning Memory for a RAM Disk

Since the minimum size of a RAM disk is usually in the order of megabytes, Level 3 external memory is the only valid memory. One option would be assign the system heap to L3 memory and simply allocate the required size using malloc(), e.g.

```
u8 *pRamDiskFrame =(u8*)malloc(nRamDiskSize*512);
```

where **nRamDiskSize** is defined in terms of 512 byte sectors.

Alternatively, whole pages of memory can be assigned using a RESERVE statement in a user modifiable section of the generated LDF. For example, the following provides enough space for 8MB of RAM disk in bank 2:

```
RESERVE(RamDiskHeap=0x02800000, RamDiskHeap_length=8M)
```

To use this values in a source module, add the following in the global definition scope,

```
extern "asm" int RamDiskHeap;
extern "asm" int RamDiskHeap_length;
```

and assign to local variables, either globally or in a function, as follows

```
u8 *pRamDiskFrame =(u8*)&RamDiskHeap;
u32 nRamDiskSize =(u32)&RamDiskHeap_length/512;
```

2.3. Getting ready for the File System Service

Since a default configuration of a RAM disk cannot be defined, the header file does not provide an `ADI_FSS_DEVICE_DEF` instance for the driver. Instead, the application must define its own ahead of including it in an `ADI_FSS_CMD_ADD_DRIVER` command passed to `adi_fss_Init()` or passing it to the FSS via a call to `adi_fss_RegisterDevice()`.

In either case, it is necessary to send the following control commands to the device driver after it has been opened with a call to `adi_dev_Open()`:

`ADI_RAMDISK_CMD_SET_RAMDISK_START_LOCATION`	The start address of the memory to use.
`ADI_RAMDISK_CMD_SET_SIZE`	The size of the RAM disk, in 512-byte sectors.

If using the `ADI_FSS_CMD_ADD_DRIVER` command, these commands must be included in the configuration table assigned to the `pConfigTable` member of the `ADI_FSS_DEVICE_DEF` structure. If opened externally of the FSS – using `adi_fss_RegisterDevice()` – it is necessary to add the following command as well:

`{ADI_DEV_CMD_SET_DATAFLOW_METHOD, (void *)ADI_DEV_MODE_CHAINED}`

Use of `adi_fss_RegisterDevice()` is detailed in Section 9.3. The `ADI_FSS_DEVICE_DEF` structure is defined in Section 7.3; the values to use are:

`DeviceNumber`	0
`pEntryPoint`	`&ADI_RAMDISK_Entrypoint`
`pConfigTable`	Pointer to configuration table as detailed above.
`pCriticalRegionData`	`NULL`
`Direction`	`ADI_DEV_DIRECTION_BIDIRECTIONAL`
`DeviceHandle`	Location to store handle of RAM Disk driver instance
`DefaultMountPoint`	An `ADI_FSS_VOLUME_IDENT` value, which is the same as a single character. E.g. `'r'`, or `0` to take next available drive letter.

However, unless the RAM disk has already been formatted, the FSS will not be able to access it since no file system can be assigned to it. A case when it will be already formatted is an embedded application where the files in RAM need to persist across system restart. The next section deals with formatting the RAM disk.

2.4. Formatting the RAM DISK

Before the RAM Disk can be assigned to the FSS it must first be formatted. To do so, requires explicit opening and configuration of the PID, creating a FAT driver instance and formatting it to your requirements. To make this simpler, the RAM Disk PID API provides an factory type function,

`ConfigRamDiskFssDeviceDef()`, to open, configure and format the RAM Disk. This function returns the handle to an adequately formed `ADI_FSS_DEVICE_DEF` structure to be passed to `adi_fss_RegisterDevice()`.

Since RAM disks will be much smaller than the maximum size for FAT16 (2GB), this function formats the RAM as FAT16 with the volume label, FAT16RAM. The minimum size of RAM disk so formatted is 4.1MB (4200 sectors). The sector size used is the usual 512 bytes.

The prototype for the `ConfigRamDiskFssDeviceDef()` function is

```
ADI_FSS_DEVICE_DEF *ConfigRamDiskFssDeviceDef(
                        u8                      *pRamDiskFrame,
                        u32                     nRamDiskSize,
                        ADI_DEV_MANAGER_HANDLE  hDevMgrHandle,
                        ADI_DCB_HANDLE          hDcbHandle,
                        u32                     nUsbDelay
);
```

And the arguments are defined in the following table:

`pRamDiskFrame`	Pointer to memory to use as RAM
`nRamDiskSize`	Size of RAM Disk array in 512 byte sectors, minimum value of 4200
`hDevMgrHandle`	Device Manager Handle – cannot be NULL
`hDcbHandle`	DCB Queue handle – can be NULL
`nUsbDelay`	Delay (microseconds) for USB connectivity. See definition of `ADI_RAMDISK_CMD_SET_USB_DELAY` command.

The function will create a temporary FAT driver instance using the minimum sizes of the FAT sector and directory entry cluster caches of 4 sectors and 2 clusters respectively.

Subsequent to this call either the `adi_fss_RegisterDevice()` function can be used to register the device with the FSS, or the driver can be closed, and the `ADI_FSS_DEVICE_DEF` structure passed to adi_fss_Init(). In the latter case, the `ADI_FSS_DEVICE_DEF` structure will need to be populated as above (only the `Direction`, `DeviceHandle`, and `DefaultMountPoint` are assigned in the call to `ConfigRamDiskFssDeviceDef()`).

Please refer to the shell_browser example for the ADSP-BF533 EZ-KIT for an example of using the `ConfigRamDiskFssDeviceDef()` and `adi_fss_RegisterDevice()` functions.

2.5. Dynamic Memory Requirements

The following table details the amount of dynamic memory required. Please refer to Section 5 below and the File System Service chapter in the "Device Drivers and System Services Manual for Blackfin® Processors" for details on assigning heaps to a PID.

Operation	FSS Cache Type	Size (bytes)
Device Instance.	General Heap	88
	TOTAL	**88**

3. Files

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

3.1. Include Files

The driver sources include the following include files:

- **`<drivers/pid/ramdisk/adi_ramdisk.h>`**
  This file contains all definitions, function prototypes etc. specific to the RAM Disk PID.

3.2. Source Files

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

- **`<Blackfin/lib/src/drivers/pid/ramdisk/adi_ramdisk.c>`**
  This file contains all the source code for the ADI RAM Disk PID. All source code is written in 'C'. There are no assembly level functions in this driver.

4. Lower Level Drivers

No lower level drivers are used.

5. Resources Required

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

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. The RAM Disk PID makes use of the general heap only as defined in the FSS for all housekeeping structures. The cache heap is not required.

Section 2.5 details the dynamic memory requirements.

5.1. Interrupts

The RAM Disk PID does not use any interrupts.

5.2. DMA

The RAM Disk PID does not use any DMA channels:

5.3. Semaphores

The RAM Disk PID requires memory for two semaphores, one to provide exclusive access and the other to use for data transfers.

5.4. Timers

The Timer service is not used by this driver.

5.5. Real-Time Clock

The RTC service is not used by this driver

5.6. Programmable Flags

No Programmable Flags are used by this driver.

5.7. Pins

No pins are used by this driver.

6. Supported Features of the Device Driver

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

6.1. Directionality

The RAM Disk PID supports the dataflow directions listed in the table below.

ADI_DEV_DIRECTION	Description
ADI_DEV_DIRECTION_BIDIRECTIONAL	Supports both the reception of data and transmission of data through the device.

Table 2 – Supported Dataflow Directions

6.2. Dataflow Methods

The RAM Disk PID can only support the `ADI_DEV_MODE_CHAINED` dataflow method. When used within the FSS this is applied automatically. If the RAM Disk PID is operated in standalone mode, or is opened prior to being registered with the FSS, then it is essential to send the following command-value pair to the driver:

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

6.3. Buffer Types

The RAM Disk PID supports the buffer types listed below.

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

6.4. Command IDs

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

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

- **DeviceHandle** – This parameter is an `ADI_DEV_DEVICE_HANDLE` type that uniquely identifies the device driver. This handle is provided to the client on return from the adi_dev_Open() function call.
- **CommandID** – This parameter is a u32 data type that specifies the command ID.
- **Value** – This parameter is a void * whose value is context sensitive to the specific command ID.

The sections below enumerate the command IDs that are supported by the driver and the meaning of the Value parameter for each command ID.

6.4.1. Device Manager Commands

The commands listed below are supported and processed directly by the Device Manager. Only the commands relevant to the RAM Disk PID are detailed.

- **ADI_DEV_CMD_TABLE**
  - o Table of command pairs being passed to the driver
  - o Value – ADI_DEV_CMD_VALUE_PAIR *

- **ADI_DEV_CMD_END**
  - o Signifies the end of a command pair table
  - o Value – ignored

- **ADI_DEV_CMD_PAIR**
  - o Single command pair being passed
  - o Value – ADI_DEV_CMD_VALUE_PAIR *

- **ADI_DEV_CMD_SET_DATAFLOW_METHOD**
  - o 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.
  - o Value – ADI_DEV_MODE enumeration

- **ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT**
  - o Determines if the device driver is supported by peripheral DMA
  - o Value – u32 * (location where TRUE or FALSE is stored). The RAM Disk PID always returns false.

- **ADI_DEV_CMD_SET_DATAFLOW**
  - o Enables/disables dataflow through the device
  - o Value – TRUE/FALSE

6.4.2. Common PID Commands

The command IDs listed below are supported and processed by the RAM Disk PID. These command IDs are unique to the File System Service

- **ADI_FSS_CMD_GET_BACKGRND_XFER_SUPPORT**
  - o Requests the RAM Disk PID to return TRUE or FALSE depending on whether the device supports the transfer of data in the background. The name of this command can be misleading, as its main purpose is to ascertain whether or not the FSS cache can be used with good effect. The RAM Disk PID currently returns TRUE, since the use of the file cache can still be advantageous.
  - o Value – Client provided location to store result.

- **ADI_FSS_CMD_GET_DATA_ELEMENT_WIDTH**
  - o Requests the RAM Disk PID to return the width (in bytes) that defines each data element. The RAM Disk PID always returns a value of 4.
  - o Value – Client provided location to store result.

- **ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE**
  - o Requests the RAM Disk PID to grant a Lock Semaphore to give the calling module exclusive access to the PID data transfer functions.
  - o Value – NULL.

- **ADI_FSS_CMD_RELEASE_LOCK_SEMAPHORE**

- o Requests the PID to release the Lock Semaphore granted in response to the `ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE` command.
  - o Value – `NULL`.

- **`ADI_FSS_CMD_SET_CACHE_HEAP_ID`**
  - o Instructs the RAM Disk PID instance to use the given Heap Index for any dynamically allocated data buffers. The default heap Index for such buffers defaults to -1, indicating that the FSS General Heap is to be used.
  - o Value – the Index of the required heap.

- **`ADI_PID_CMD_GET_FIXED`**
  - o Requests the RAM Disk PID to return `TRUE` or `FALSE` depending on whether the device is to be regarded as Fixed or removable. The RAM Disk PID returns `FALSE`.
  - o Value – Client provided location to store result.

- **`ADI_PID_CMD_MEDIA_ACTIVATE`**
  - o Activates the RAM Disk PID, configuring it for use.
  - o Value – `TRUE` to activate, `FALSE` to deactivate.

- **`ADI_PID_CMD_POLL_MEDIA_CHANGE`**
  - o Instructs the RAM Disk PID to check the status of the device for the removal or insertion of media. If the driver detects that media has been removed it issues the `ADI_FSS_EVENT_MEDIA_REMOVED` callback event to the Device Manager Callback function. If the driver detects that media has been inserted it issues the `ADI_FSS_EVENT_MEDIA_INSERTED` callback event, (Section 6.5.2).
  - o Value – `NULL`.

- **`ADI_PID_CMD_DETECT_VOLUMES`**
  - o Instructs the RAM Disk PID to discover the volumes/partitions available on the media. For each volume detected the PID issues the `ADI_FSS_EVENT_VOLUME_DETECTED` event, passing the pointer to the salient volume information as the third argument. (See Section 6.5.2).
  - o Value – `NULL`.

- **`ADI_PID_CMD_GET_MAX_TFRCOUNT`**
  - o Requests the maximum size, in bytes, of a single transfer (LBA request) . The RAM Disk PID is only limited by the size of the `ElementCount` member of the `ADI_DEV_1D_BUFFER` structure, which is declared as `u32`, and so it always returns a value of `0xFFFFFFFF`.
  - o Value – Client provided location to store result.

- **`ADI_PID_CMD_SEND_LBA_REQUEST`**
  - o Requests the RAM Disk PID to command the device to read/write a number of sectors from/to a given LBA start sector.
  - o Value – Address of the `ADI_FSS_LBA_REQUEST` structure containing the above information. This is ignored by the RAM Disk PID driver since the LBA information is communicated to the driver as part of the ADI_FSS_SUPER_BUFFER structure passed to the `adi_dev_Read/Write()` functions.

- **`ADI_PID_CMD_ENABLE_DATAFLOW`**
  - o Instructs the RAM Disk PID to take the necessary steps to begin/stop dataflow. This is ignored by the RAM Disk PID driver since all data transfer has completed on return from the `adi_dev_Read/Write()` functions.
  - o Value - `TRUE`/`FALSE`.

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

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

    ADI_FSS_VOLUME_DEF fields

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

The default values for the RAM Disk PID are a sector size of 512 bytes starting at logical sector 0.

6.4.3. RAM Disk PID Specific Commands

The command IDs listed below are unique to the RAM Disk PID.

- **ADI_RAMDISK_CMD_SET_RAMDISK_START_LOCATION:**
  - o Defines the memory area to use as RAM Disk.
  - o Value – the address of the L3 memory area to be used for the RAM disk.
  - o Default – none.

- **ADI_RAMDISK_CMD_SET_SIZE**
  - o Defines the size of the RAM Disk.
  - o Value - the size in 512 byte sectors.
  - o Default – NOne.

- **ADI_RAMDISK_CMD_SET_USB_DELAY**
  - o Sets the delay in microseconds to be inserted after completion of each transfer. This is sometimes required to enable the driver to be used with the USB MSC device class.
  - o Value – delay in microseconds.
  - o Default – 0.

6.5. Callback Events

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

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

- `ClientHandle`. Except for callbacks to the direct callback function this `void*` parameter will be the `DeviceHandle` (3rd) argument passed to the `adi_pdd_Open` function of the PID. For direct callbacks it must be the address of this argument.

- `EventID`
  This is a `u32` data type that specifies the event ID. See below.

- `Value`
  This parameter is a `void*` whose value is context sensitive to the specific event ID.

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

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

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

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

6.5.1. Common Events

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

- **ADI_DEV_EVENT_BUFFER_PROCESSED**
  Notifies callback function that a chained I/O buffer has been processed by the device driver.
  Value – This value is the `CallbackParameter` value that was supplied in the buffer that was passed to the `adi_dev_Read()` or `adi_dev_Write()` function. For compliance with the FSS requirements this value is the address of this buffer.

6.5.2. FSS Specific Events

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

- **ADI_FSS_EVENT_MEDIA_INSERTED**
  This event is issued in response to the `ADI_PID_CMD_POLL_MEDIA_CHANGE` command upon detection that both media is present and a valid FTL license has been detected.
  Value – The address of a data location. On issue of the callback this location contains the Device Number of the device (zero). On return from the callback the location contains a result code. If the result code returned is `ADI_FSS_RESULT_SUCCESS`, the RAM Disk PID will regard the media as being present and correctly accounted for by the FSS.

- **ADI_FSS_EVENT_VOLUME_DETECTED**
  This event is issued in response to the `ADI_PID_CMD_DETECT_VOLUMES` command upon detection of a valid volume/partition.
  Value – The address of an `ADI_FSS_VOLUME_DEF` structure defining the volume:

`FileSystemType`	-	The File system type, as defined in the adi_fss.h header file under the title "Enumerator for known File System types". See the FSS Implementation document for further details.
`StartAddress`	-	The Sector (LBA value) of the first sector in the volume. This is always zero of the RAM Disk PID.
`VolumeSize`	-	The size of the volume in sectors.
`SectorSize`	-	The size in bytes of each sector on the volume. The default size is 512 bytes.
`DeviceNumber`	-	The number of the device in a chain of devices. This is not applicable and is set to zero.

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

- **ADI_PID_EVENT_DEVICE_INTERRUPT**
  This event is issued by the RAM Disk PID once all data pertaining to an LBA request is processed.
  Value – The address of the Buffer structure associated with the event. This must be the value located in the `pBuffer` field of the associated LBA request structure.

6.6. Return Codes

All API functions of the RAM Disk PID return a status code indicating either successful completion of the function or an indication that an error has occurred. This section enumerates the return codes that the device driver is capable of returning to the client. A return value of `ADI_DEV_RESULT_SUCCESS` or `ADI_FSS_RESULT_SUCCESS` indicates success, while any other value indicates an error or some other informative result. The values `ADI_DEV_RESULT_SUCCESS` and `ADI_FSS_RESULT_SUCCESS` are always equal to the value zero. All other return codes are a non-zero value.

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

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

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

6.6.1. Common Return Codes

The return codes described in this section are common to many device drivers. The list below enumerates all common return codes that are supported by the RAM Disk PID.

- **ADI_DEV_RESULT_SUCCESS**
  The function executed successfully.

- **ADI_DEV_RESULT_NOT_SUPPORTED**
  The function is not supported by the driver.

- **ADI_DEV_RESULT_DEVICE_IN_USE**
  The requested device is already in use.

- **ADI_DEV_RESULT_NO_MEMORY**
  There is insufficient memory available.

- **ADI_DEV_RESULT_BAD_DEVICE_NUMBER**
  The device number is invalid.

- **ADI_DEV_RESULT_DIRECTION_NOT_SUPPORTED**
  The device cannot be opened in the direction specified.

- **ADI_DEV_RESULT_BAD_DEVICE_HANDLE**
  The handle to the device driver is invalid.

- **ADI_DEV_RESULT_BAD_MANAGER_HANDLE**
  The handle to the Device Manager is invalid.

- **ADI_DEV_RESULT_BAD_PDD_HANDLE**
  The handle to the physical driver is invalid.

- **ADI_DEV_RESULT_INVALID_SEQUENCE**
  The action requested is not within a valid sequence.

- **ADI_DEV_RESULT_ATTEMPTED_READ_ON_OUTBOUND_DEVICE**
  The client attempted to provide an inbound buffer for a device opened for outbound traffic only.

- **ADI_DEV_RESULT_ATTEMPTED_WRITE_ON_INBOUND_DEVICE**
  The client attempted to provide an outbound buffer for a device opened for inbound traffic only.

- **ADI_DEV_RESULT_DATAFLOW_UNDEFINED**
  The dataflow method has not yet been declared.

- **ADI_DEV_RESULT_DATAFLOW_INCOMPATIBLE**
  The dataflow method is incompatible with the action requested.

- **ADI_DEV_RESULT_BUFFER_TYPE_INCOMPATIBLE**
  The device does not support the buffer type provided.

- **ADI_DEV_RESULT_NON_TERMINATED_LIST**
  The chain of buffers provided is not NULL terminated.

- **ADI_DEV_RESULT_NO_CALLBACK_FUNCTION_SUPPLIED**
  No callback function was supplied when it was required.

- **ADI_DEV_RESULT_REQUIRES_BIDIRECTIONAL_DEVICE**
  Requires the device be opened for bidirectional traffic only.

6.6.2. FSS Specific Return codes used by the RAM Disk PID driver

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

- **ADI_FSS_RESULT_NO_MEDIA**
  No media is detected, or no valid FTL license detected.

- **ADI_FSS_RESULT_NO_MEMORY**
  There was insufficient memory to complete a request. Usually as a result of a call to
  _adi_fss_malloc().

- **ADI_FSS_RESULT_FAILED**
  General failure.

- **ADI_FSS_RESULT_NOT_SUPPORTED**
  The requested operation is not supported by the PID.

- **ADI_FSS_RESULT_SUCCESS**
  General Success.

7. Data structures

7.1. Device Driver Entry Points, ADI_DEV_PDD_ENTRY_POINT

This structure is used in common with all drivers that conform to the ADI Device Driver model, to define
the entry points for the device driver. It is defined in the RAM Disk PID source module, adi_ramdisk.c, and
declared as an extern variable in the RAM Disk PID header file, adi_ramdisk.h:

```
extern ADI_DEV_PDD_ENTRY_POINT ADI_RAMDISK_Entrypoint;
```

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 RAM Disk 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 of command-value pairs is declared in the RAM Disk PID header file, adi_ramdisk.h.

7.3. Device Definition Structure, ADI_FSS_DEVICE_DEF

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

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

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

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

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

```
#if !defined(__ADI_RAMDISK_HOST_C__)
:
#if defined(_ADI_RAMDISK_DEFAULT_DEF_)
static ADI_FSS_DEVICE_DEF ADI_RAMDISK_Def = { … };
:
#endif
:
#endif
```

7.4. Volume Definition Structure, ADI_FSS_VOLUME_DEF

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

```
typedef struct {
    u32 FileSystemType;
    u32 StartAddress;
    u32 VolumeSize;
    u32 SectorSize;
    u32 DeviceNumber;
} ADI_FSS_VOLUME_DEF;
```

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

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

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

7.5. The FSS Super Buffer Structure, ADI_FSS_SUPER_BUFFER

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

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 de-referenced and its contents made use of.

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

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

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

The definition of the structure is:

```
typedef struct ADI_FSS_SUPER_BUFFER{
    ADI_DEV_1D_BUFFER      Buffer;
    struct adi_cache_block  *pBlock;
```

```
    u8                      LastInProcessFlag;
    ADI_FSS_LBA_REQUEST     LBARequest;
    ADI_SEM_HANDLE          SemaphoreHandle;
    ADI_FSS_FILE_DESCRIPTOR *pFileDesc;
    ADI_DCB_CALLBACK_FN     FSDCallbackFunction;
    void                    *FSDCallbackHandle;
    ADI_DCB_CALLBACK_FN     PIDCallbackFunction;
    void                    *PIDCallbackHandle;
} ADI_FSS_SUPER_BUFFER;
```

Where the fields are defined as:

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

7.6. LBA Request, ADI_FSS_LBA_REQUEST

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

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

```
} ADI_FSS_LBA_REQUEST;
```

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

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

8. Data Transfer

All data transfer is initiated and completed within each call to `adi_dev_Read()` or `adi_dev_Write()` with the RAM Disk PID device handle. After each sub-buffer is processed the `ADI_DEV_EVENT_BUFFER_PROCESSED` event is issued via the usual Device Manager callback function. Upon completion of all buffers in the chain, the `ADI_PID_EVENT_DEVICE_INTERRUPT` event is issued.

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

These callbacks are made via the Device Manager with the following arguments:

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

2   The appropriate event code.

3   The address of the `ADI_FSS_SUPER_BUFFER` structure for the sub-buffer just completed.

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

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

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

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

1. Acquire Lock Semaphore from the RAM Disk PID passing the command-value pair,
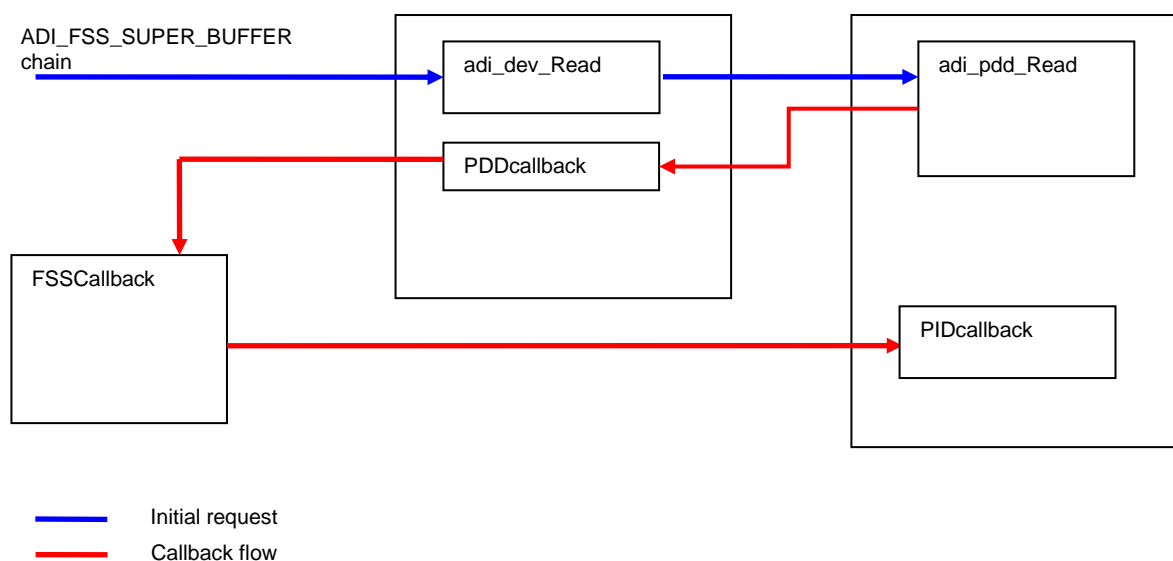
```
{ ADI_FSS_CMD_ACQUIRE_LOCK_SEMAPHORE, NULL },
```

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

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

The LBA request member of the `ADI_FSS_SUPER_BUFFER` structure chain at location `pSuperBuffer` must be correctly assigned for each buffer in the chain. All data transfer will have completed upon return from this call.

3. The Lock Semaphore acquired in stage 1 is released by the FSD/application either upon completion of receipt of the `ADI_PID_EVENT_DEVICE_INTERRUPT` event.

The diagram below illustrates the command and callback flow of the RAM Disk PID.

9. Initialization of the RAM Disk Physical Interface Driver

This section describes the initialization stages required for the RAM DiskPID. How the RAM Disk PID is initialized will depend largely on how it is to be used. Possible usage scenarios are:

- Permanently assigned to the FSS

- Shared use between the FSS and another function. For example, the RAM Disk may be connected to a host computer via USB to download media files that are subsequently played by the embedded application via the FSS once the cable is removed.

- Standalone

In all cases the RAM Disk PID is required to undergo certain stages of configuration before it can be used by an application; the differences between the above scenarios concerns which part of the software (application vs library) is responsible for what. The permanent assignment to the FSS requires the least amount of effort by the application; with the standalone case requiring the most effort.

Before showing how to prepare the RAM Disk PID in each scenario we will describe each of the stages of configuration.

9.1. Procedure for Opening the RAM Disk PID

The configuration stages of RAM Disk PID are

- Open
- Configuration
- Activation
- Poll for Media

Once activated the RAM Disk PID is ready for use. Conversely to tear down the RAM Disk PID requires the following stages to be performed:

- Deactivation
- Close

9.1.1. Open Stage

In this stage the RAM Disk PID is opened in the usual way for a device driver conforming to the device driver model. The device driver definition structure, `ADI_RAMDISK_Def`, (Section 7.3) provides most of the requirements for the call to `adi_dev_Open()` to open the RAM Disk PID device driver:

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

The other arguments need to be supplied: The `<DeviceManagerHandle>` and `<DMAManagerHandle>` are those obtained from the usual initialization of the System Services & Device Manager. The `<DCBQueueHandle>` is the handle of the DCB queue if callbacks to `<Callback-function>` from the RAM Disk PID are to be deferred, (recommended). The `ADI_RAMDISK_Def` structure is provided in the RAM Disk PID header file and is available to the application if the `_ADI_RAMDISK_DEFAULT_DEF_` macro is defined ahead of the include statement:

```
#define _ADI_RAMDISK_DEFAULT_DEF_
#include <drivers/pid/usb/adi_ramdisk.h>
```

Alternatively, the following values can be used:

adi_dev_Open argument	Value
pEntryPoint	&ADI_RAMDISK_Entrypoint
DevNumber	0
Direction	ADI_DEV_DIRECTION_BIDIRECTIONAL

The application is required to perform the Open Stage only in the standalone and shared scenarios; the FSS makes use of the `ADI_RAMDISK_Def` structure to open the driver internally.

To close the RAM Disk PID, simply call `adi_dev_Close()` with the RAM Disk PID device handle, e.g:

```
adi_dev_Close( ADI_RAMDISK_Def.DeviceHandle );
```

9.1.2. Configuration Stage

In this stage command value pairs are passed to the RAM Disk PID via calls to adi_dev_Control() .

The default command-value pairs are defined in the `ADI_RAMDISK_ConfigurationTable[]` table defined in the RAM Disk PID header file. The address of this assigned to the `pConfigTable` member of the `ADI_RAMDISK_Def` structure also defined in the header file.

Optional commands are required to be overridden. Example commands would be those required to change the size of the reserved area at the start of the RAM DISK and to specify the RAM DISK information details.

The mandatory commands are given in Section 9.1.2.2. Of these the `ADI_DEV_CMD_SET_DATAFLOW_METHOD` command is only required in the standalone and shared scenarios; the FSS always issues this command internally. The commands required for the default behavior of the RAM Disk PID are detailed in the RAM Disk PID header file by a table of found in Sections 6.4.2 and 6.4.3.

9.1.2.1. Default Settings

The RAM Disk PID has no default settings.

9.1.2.2. Additional Required Configuration Settings

In addition to the possible overrides of the default driver settings, the RAM Disk PID responds to the following commands issued as detailed below. The following table does not itemize the mandatory commands required to communicate to the RAM Disk PID Driver (see Section 6.4 for further details).

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

Table 3 – Additional Required Settings

9.1.3. Activation Stage

The Activation Stage is entered by sending the `ADI_PID_CMD_MEDIA_ACTIVATE` command to the RAM Disk PID with the associated value as `true`. Similarly it is exited with the same command but with the associated value as `false`.

On entering this stage all GPIO pin assignments are made, the flash translation layer is activated and the PID is ready to receive requests to transfer data to and from the RAM Disk Device[1].

9.1.4. Poll for Media Stage

At this point the RAM Disk PID should be polled for the detection of media. Depending on whether the file system is to be accessed or whether direct access is required, this stage is done in one or two parts.

The first part is to send the `ADI_PID_CMD_POLL_MEDIA_CHANGE` command which results in a callback with the `ADI_FSS_EVENT_MEDIA_INSERTED` event. This event is sent to the callback function specified in the open stage - via the deferred callback manager if also specified in the open stage - or it can be sent directly to the same or a different callback function by passing the required function address with the `ADI_PID_CMD_SET_DIRECT_CALLBACK` command. It is imperative to do the latter if the second part of this stage is to be performed from within the callback function. For both usage scenarios involving the FSS this is done internally by the FSS.

This callback uses the following arguments:

   o   The address of a location containing the RAM Disk PID device handle.

   o   The `ADI_FSS_EVENT_MEDIA_INSERTED` event.

   o   The address of a u32 variable. The contents of this variable are set by the RAM Disk PID to the Device Number of the device on the chain for which media is detected, if appropriate. The callback function shouid set this variable to an appropriate result code, either `ADI_FSS_RESULT_FAILED` or `ADI_FSS_RESULT_SUCCESS`, the latter value indicating the detected media has been dealt with accordingly.

The second part of this stage is dependent on whether the media has been formatted with a file system. This is often only relevant when used in association with the FSS and as such is done internally. The detection of a formatted partition (called volume) on the RAM DISK is performed by sending the

`ADI_PID_CMD_DETECT_VOLUMES` command to the RAM Disk PID. If a volume is detected the callback function is invoked with the following arguments

- o The address of a location containing the the RAM Disk PID device handle.

- o The `ADI_FSS_EVENT_VOLUME_DETECTED` event.

- o The address of an `ADI_FSS_VOLUME_DEF` structure detailing the volume found.

9.2. Assigning the RAM Disk PID Permanently to the FSS

Having described in detail each of the stages involved, we can turn to the business of how these stages are performed in relation to each of the usage scenarios. In particular we emphasize here the responsibilities of the application developer.

The simplest of these scenarios is the exclusive use of the RAM Disk PID by the FSS. Section 2.2 detailed what is required by the application to assign the RAM Disk PID to the FSS with its default settings. In this section we will show what is required to alter these defaults. Examples of when this may be applicable are if a different RAM Disk Device is to be used; or a different reserved area size is required.

In these cases a separate instance of the `ADI_FSS_DEVICE_DEF` structure must be defined (Section 7.3). If the same name of default `ADI_FSS_DEVICE_DEF` structure is used it is important to ensure that the `_ADI_RAMDISK_DEFAULT_DEF_` macro is undefined ahead of the RAM Disk PID header include statement.

This structure will require the address of the RAM Disk PID entry point structure, `ADI_RAMDISK_EntryPoint` (Section 7.1), and the address of a configuration table made up of command-value pairs (Section 7.2) . The entry point structure and a default configuration table are defined in the RAM Disk PID header file, <drivers/pid/ramdisk/adi_ramdisk.h>. An example configuration table and definition structure could be:

```
ADI_DEV_CMD_VALUE_PAIR ADI_RAMDISK_ConfigTable [] = {
        { ADI_RAMDISK_CMD_SET_RAM DISK_INFO,           (void *)&ADI_EZKIT_RAM
DISK_Info },
        { ADI_RAMDISK_CMD_SET_RESERVED_SIZE,      (void *)10 },
        { ADI_DEV_CMD_END,                   NULL },
};


ADI_FSS_DEVICE_DEF ADI_RAMDISK_Def = {
        0,
        &ADI_RAMDISK_EntryPoint,
        ADI_RAMDISK_ConfigTable,
        NULL,
        ADI_DEV_DIRECTION_BIDIRECTIONAL,
        'n'
};
```

Please note that the reserved size must always be the same as that used for formatting; the default value is 10 blocks. Please also note that the FSS will endeavor to apply the specified default mount point drive letter to this device. 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".

Once the `ADI_FSS_DEVICE_DEF` structure is defined it is simply assigned to the FSS with the following command-value pair added to the FSS configuration table passed to adi_fss_Init():

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

The FSS will then perform of all the stages of device initialization as described above in Section 9.1.

9.3. Registering the RAM Disk PID with the File System Service

If the RAM Disk PID is intended to be used temporarily in an application or it is required to switch between the FSS and the USB MSD, then a different procedure is required to register it with the FSS. This approach requires the application developer to manage the *open* and *configuration* stages of the device initialization, leaving the activation and media polling stages to the FSS. Once configured the RAM Disk PID can be registered with the FSS at any time after a the call to `adi_fss_Init()` using either the `adi_fss_RegisterDevice()` function, .e.g:

        adi_fss_RegisterDevice(&ADI_RAMDISK_Def, <poll-flag> };

Alternatively, registration can be effected by passing the following command value pair to the driver via **adi_dev_Control()**:

        { ADI_FSS_CMD_REGISTER_DEVICE, (void*)&ADI_RAMDISK_Def },

The `<poll-flag>` argument in the `adi_fss_RegisterDevice()` function is used to determine whether the RAM Disk PID is to be polled for media immediately upon registration. A value of true will effect the immediate polling and a value of false is synonymous with the usage of the command value pair.

The information used in the default `ADI_FSS_DEVICE_DEF` structure can be utilized for the opening of the device (see Section 9.1.1), and the default configuration table can be used to configure the device.

9.4. Registering the RAM Disk PID with the USB Mass Storage Class Driver

When the RAM Disk PID is to be used as a USB mass storage device, the same stages apply as for when it is registered with the FSS. However, in this case a call to the USB mass storage class driver is made instead; the RAM Disk PID device handle is sent to the class driver via a call to `adi_dev_Control()` with the `ADI_USB_MSD_CMD_REGISTER_FSSPID` command:

`{ ADI_USB_MSD_CMD_REGISTER_FSSPID, (void*) *)ADI_RAMDISK_Def.DeviceHandle },`

The above command should be used ahead of the `ADI_USB_MSD_CMD_SCSI_INIT` command.

9.5. Initialization when used standalone

The final usage scenario is where the RAM Disk PID is to be used outside the context of the FSS or the USB mass storage class driver. In this case, all of the initialization stages with the exception of the volume detection part of the *Poll for Media* stage must be performed by the application developer.

A callback function will have to be defined to handle the following events. In all these events the first argument in the callback is the address of a location containing the RAM Disk PID Device Handle, the Event will be one of the following and the third argument is interpreted as required, and detailed below.

1. **ADI_FSS_EVENT_MEDIA_INSERTED**. The third argument is the address of a location containing the device number (`0`) of the device for which media is detected. On return it must contain a result code, indicating whether the callback has been handled successfully.

2. **ADI_FSS_EVENT_MEDIA_REMOVED**. The third argument has no meaning in this event. The action to take will depend on the purpose of the application.

3. **ADI_FSS_EVENT_VOLUME_DETECTED**. The third argument is the address of an `ADI_FSS_VOLUME_DEF` structure defining the volume. Please refer to Section 7.4 for details about the definition and assignment of this structure. The action to take will depend on the purpose of the application.

4   **ADI_DEV_EVENT_BUFFER_PROCESSED**. This is the host transfer completion event. Please refer to Section 8 for further details.

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

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

An example of standalone use can be observed in the so-called Raw PID access functions detailed in the following module in the VisualDSP++ 5.0 installation:

```
Blackfin\lib\src\drivers\pid\adi_rawpid.c
```