Media Driver Interface Guide

Version 1.40

For use with Media Driver Base versions 1.03 and above

Exported on 02/22/2019

Table of Contents

1 System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

- Introduction – describes the main elements of the module.
- Feature Check – summarizes the main features of the module as bullet points.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

Note: To download this manual as a PDF, see File System Media Driver PDFs.

1.1 Introduction

This document defines HCC's Media Driver Interface specification. Defining a standard interface means that media drivers can be written independently of any specific file system implementation. All HCC's FAT-based file systems use this specification. It is designed in a flexible way so that, regardless of how a file system expects its media driver to be structured, you can easily create a wrapper to use HCC's Media Driver Interface.

Media drivers provide an interface for a file system to read and write physical media. Typical examples of media drivers include the following:

- A RAM drive.
- An MMC/SD card interface.
- A SCSI for interfacing to a USB mass storage device (for example, a pen drive).
- A Flash Translation Layer (FTL) for accessing flash memory devices.

The Media Driver Interface specification allows a single media driver to support one or more physical media, each of these being represented as a different drive at the media driver interface. The file system handles all drives identically, regardless of their internal design features, though these may vary widely.

The diagram below shows a typical system architecture including a file system and media drivers.
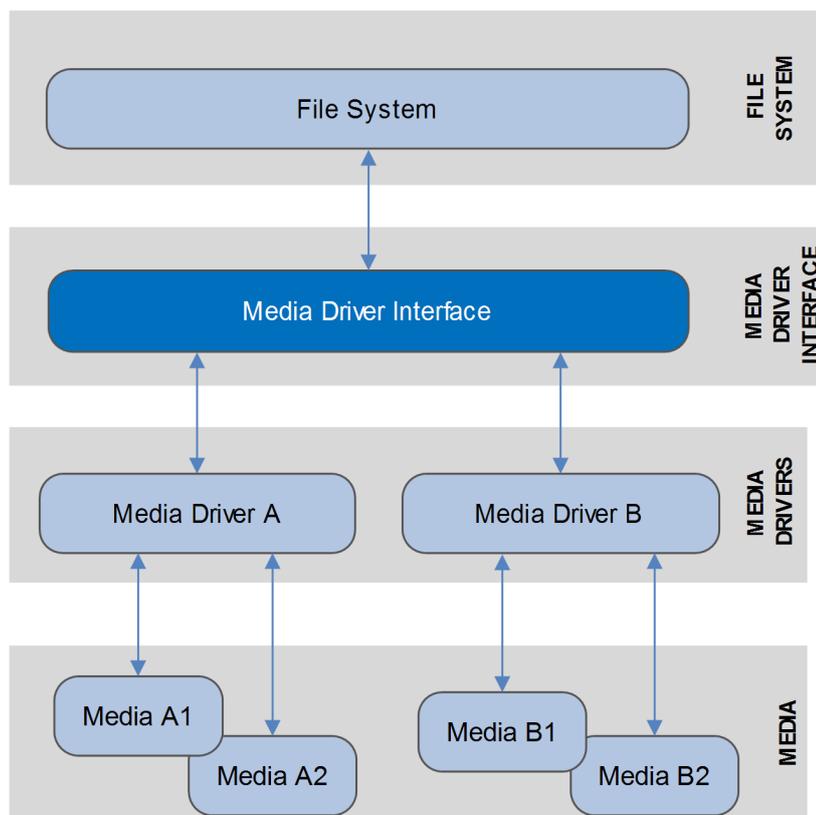
This document specifies the common interface that each of the media drivers must conform to.

Typically a media driver handles a class of drives with common attributes. For example, an SD card media driver would handle multiple SD cards. (Note, however, that the driver would also be dependent on the

particular hardware controller used on the target.) Similarly, SafeFTL is a media driver for all flash devices, under which different flash drives may be created.

1.2 Feature Check

The main features of the Media Driver Interface are the following:

- A standard interface for connecting different media types to a system.
- Works with all HCC FAT-based file systems.
- Supports large block read and write.
- Provides an IOCTL capability.

1.3 Packages and Documents

Packages

The table below lists the packages that you need in order to use this module:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
media_drv_base	The base media driver package that includes the framework that all media drivers use.
psp_template_base	The base Platform Support Package (PSP).

The specific media driver packages that conform to the *HCC Media Driver Interface Specification* are named **media_drv_xxx**.

Documents

For an overview of HCC file systems and data storage, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC Media Driver Interface Guide

This is this document. Each media driver has its own user guide.

1.4 Change History

This section describes past changes to this manual.

- To download this manual or a PDF describing an earlier software version, see File System Media Driver PDFs.
- For the history of changes made to the package code itself, see History: media_drv_base.

The current version of this manual is 1.40. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
1.40	2019-02-22	1.03	New template.
1.30	2018-05-08	1.03	Added F_IOCTL_MSG_FLUSH to *F_IOCTL* section.
1.20	2017-08-18	1.01	Updated *Packages* list.
1.10	2017-01-22	1.01	*New Change History* format.
1.09	2017-01-18	1.01	Added function group descriptions to API.
1.08	2016-01-05	1.01	Added *Change History* section.
1.07	2014-08-14	1.01	Reorganized *System Overview*.
1.06	2014-05-06	1.00	Various small changes.
1.05	2014-02-07	1.00	Various small changes.
1.04	2013-12-16	1.00	First online version.

2 Source File List

This section describes all the source code files included in the media driver base system, and how files are organized in the associated media drivers. These files follow the HCC Embedded standard source tree system, described in the HCC Source Tree Guide. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

2.1 API Header File

The Application Programming Interface (API) file for the media driver base system is named **src/api/api_mdriver.h**.

This is the only file that should be included by an application using this module. For details of the single API function, see Application Programming Interface.

2.2 System Files

All conforming media drivers have the same file organization, as follows:

- The API file for the media driver is **src/api/api_mdriver_[drv_name].h**.
- The configuration file for the media driver is **src/config/config_mdriver_[drv_name].h**.
- Source files are in **src/media-drv/[drv_name]**, where **drv_name** represents the type of media driver, for example "ram".

For example, the standard HCC media driver for RAM drives has the following file organization:

File	Description
src/api/api_mdriver_ram.h	This file contains the RAM media driver external interface: the **F_DRIVERINIT()** function. This is **ram_initfunc()** in this case.
src/config/config_mdriver_ram.h	This file contains the RAM media driver configuration parameters.
src/media-drv/ram	This folder contains the RAM media driver source code.
src/version/ver_mdriver_ram.h	This file contains the version number of the RAM media driver.

2.3 Version File

The file **src/version/ver_mdriver.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

3 Application Programming Interface

The media driver is designed so that a file system can dynamically attach and detach any media driver that conforms to the HCC specification.

When a media driver is used:

1. The file system calls that media driver's **F_DRIVERINIT()** function.

2. **F_DRIVERINIT()** returns a pointer to a structure containing a set of functions for accessing the media driver.

3. The various **READ** and **WRITE** functions may be used.

4. The **F_RELEASE()** function is called to remove the specified drive, freeing any resources associated with it.

Each function is documented in detail in the following sections.

3.1 Functions

The functions are the following:

Function	Description
F_DRIVERINIT()	Initializes the interface with the driver.
F_GETPHY()	Gets the physical properties of a drive.
F_READSECTOR()	Reads a single sector.
F_READMULTIPLESECTOR()	Reads a series of consecutive sectors.
F_WRITESECTOR()	Writes a complete sector.
F_WRITEMULTIPLESECTOR()	Writes a series of consecutive sectors.
F_GETSTATUS()	Checks the status of the media.
F_RELEASE()	Removes a specified drive.
F_IOCTL()	Sends special or Out of Band (OOB) messages to the driver.

F_DRIVERINIT

Use this function to initialize the interface with the driver.

This function is called by any system that wants to use a media driver that conforms to the HCC Media Driver Interface specification. The caller passes a parameter to the initialization function of a conforming driver (for example, **ram_initfunc()**) and the driver returns a pointer to a structure defining the interface to that driver.

The parameter passed to the function is typically used to identify different instances (physical drives) for that driver to access.

Format

```
F_DRIVER * ( * F_DRIVERINIT )( unsigned long driver_param )
```

Arguments

Argument	Description	Type
driver_param	Driver parameter, typically used to identify a particular physical drive. This is normally set to 0 if the driver handles just a single medium. If you set this to F_AUTO_ASSIGN, the driver automatically assigns a free driver.	unsigned long

Return values

Return value	Description
F_DRIVER *	A pointer to the driver structure, or NULL if the request failed.

Note: An **xxx_initfunc()** must allocate or use a static structure for the F_DRIVER structure. It must return a pointer to this structure, which must contain all the driver entry points and also other data as required.

F_GETPHY

Use this function to discover the physical properties of a drive.

Format

```
typedef int ( * F_GETPHY )(
    F_DRIVER *   driver,
    F_PHY *      phy )
```

Arguments

Argument	Description	Type
driver	A pointer to the driver structure of the driver to query.	F_DRIVER *
phy	A pointer to the physical properties structure for the driver.	F_PHY *

Return values

Return value	Description
0	Successful execution.
Else	Error codes for this device. For example, device not present.

F_READSECTOR

Use this function to read a single sector.

Format

```
typedef int ( * F_READSECTOR )(
    F_DRIVER *      driver,
    void *          data,
    unsigned long   sector )
```

Arguments

Argument	Description	Type
driver	A pointer to the driver structure of the drive to read from.	F_DRIVER *
data	Where the driver should write the data.	void *
sector	The number of the sector to read.	unsigned long

Return values

Return value	Description
0	Successful execution.
Else	Sector out of range.

F_READMULTIPLESECTOR

Use this function to read a series of consecutive sectors.

> **Note:** This function is optional but its inclusion enhances performance on most devices.

Format

```
typedef int ( * F_READMULTIPLESECTOR )(
    F_DRIVER *      driver,
    void *          data,
    unsigned long   sector,
    int             cnt )
```

Arguments

Argument	Description	Type
driver	A pointer to the driver structure of the drive to read from.	F_DRIVER *
data	Where to write the data.	void *
sector	The number of the first sector to be read.	unsigned long
cnt	The number of sectors to read.	int

Return values

Return value	Description
0	Successful execution.
Else	Sector out of range.

F_WRITESECTOR

Use this function to write a complete sector.

> **Note**. This function may be omitted on read-only drives.

Format

```
typedef int ( * F_WRITESECTOR )(
    F_DRIVER *      driver,
    void *          data,
    unsigned long   sector )
```

Arguments

Argument	Description	Type
driver	A pointer to the driver structure of the drive.	F_DRIVER *
data	A pointer to the data to be written to the specified sector.	void *
sector	The number of the sector to write.	unsigned long

Return values

Return value	Description
0	Successful execution.
Else	Sector out of range.

F_WRITEMULTIPLESECTOR

Use this function to write a series of consecutive sectors.

> **Note**:
> - This function is optional, but its inclusion will enhance performance on most devices and is particularly important for hard disk drives.
> - This function may be omitted if the system contains only read-only drives.

Format

```
typedef int ( * F_WRITEMULTIPLESECTOR )(
    F_DRIVER *      driver,
    void *          data,
    unsigned long   sector,
    int             cnt )
```

Arguments

Argument	Description	Type
driver	A pointer to the driver structure of the drive to write to.	F_DRIVER *
data	A pointer to the data to be written.	void *
sector	The number of the first sector to write.	unsigned long
cnt	The number of sectors to write.	int

Return values

Return value	Description
0	Successful execution.
Else	Sector out of range.

F_GETSTATUS

Use this function to check the status of the media.

The function returns a F_ST_XXX bit field of new status information. Use this function with removable media to check that a card has not been removed or swapped, and is not read-only.

> **Note**: If the drive is a permanent medium (for example, hard disk or RAM drive), this function may be omitted.

Format

```
typedef long ( * F_GETSTATUS )( F_DRIVER * driver )
```

Arguments

Argument	Description	Type
driver	A pointer to the driver structure of the drive to check.	F_DRIVER *

Return values

Return value	Description
0	Drive is in a usable state.
Else	Status as defined in F_ST_XXX.

F_RELEASE

Use this function to remove a specified drive.

The driver can use this call to free any resources associated with that drive.

> **Note:** Use of this routine in the driver is optional.

Format

```
typedef void ( * F_RELEASE )( F_DRIVER * driver )
```

Arguments

Argument	Description	Type
driver	A pointer to the driver structure of the drive to remove.	F_DRIVER *

Return values

Return value
None.

F_IOCTL

Use this function to send special or Out of Band (OOB) messages to the driver.

> **Note:** Use of this routine in the driver is optional. The mechanism is flexible; you can add your own driver-specific messages without affecting the Media Driver Interface design.

Specify the driver to send the message to and a message descriptor to tell the driver how to handle the input and output parameters.

You can also optionally provide a pointer to a set of input parameters for the driver to handle, and a set of output parameters for the driver to return. The provision of input and output parameters is optional and defined by the message type. By using a unique message type, you can send almost any message to the driver and get a response.

The following messages are currently supported by the system:

- **END OF DELETE** – message F_IOCTL_MSG_ENDOFDELETE.
  There are no parameters to this IOCTL. This signals to the driver that a delete operation is completed. For example in HCC's FTL, all log blocks are flushed at this time.
- MULTIPLE SECTOR ERASE – tells the driver to erase the sectors which are listed.
  iparam is a structure that contains the erase information, defined by the structure: ST_IOCTL_MULTIPLESECTORERASE. *oparam* is not used.
- **F_IOCTL_MSG_FLUSH** – tells the driver to flush buffers on a remote device.

Format

```
typedef int ( *F_IOCTL )(
    F_DRIVER *      driver,
    unsigned long   msg,
    void *          iparam,
    void *          oparam )
```

Arguments

Argument	Description	Type
driver	A pointer to the driver structure of the driver.	F_DRIVER *
msg	The message type being sent.	unsigned long
iparam	A pointer to parameters for the driver to receive.	void *
oparam	A pointer to parameters for the driver to return.	void *

Return values

Return value	Description
0	Successful execution.
Else	Failure.

3.2 Types and Definitions

This section describes the main elements that are defined in the API Header file.

F_PHY

This is the format of the *F_PHY* structure:

Element	Type	Description
number_of_cylinders	unsigned short	Not normally used, this is available for historical reasons.
sector_per_track	unsigned short	Not normally used, this is available for historical reasons.
number_of_heads	unsigned short	Not normally used, this is available for historical reasons.
number_of_sectors	unsigned long	The number of physical sectors available on the media.
media_descriptor	unsigned char	This takes one of the F_MEDIADESC_XXX values.
bytes_per_sector	unsigned short	The number of bytes in each sector.

Media Type Definitions

There are just two definitions:

Type	Description
F_MEDIADESC_REMOVABLE	The attached media is removable.
F_MEDIADESC_FIX	The attached media is not removable.

Multisector Erase IOCTL Input Parameter Structure

This is the format of the structure:

Element	Type	Description
one_sector_databuffer	void *	A pointer to a buffer containing erase data for a single sector.
start_sector	unsigned long	The number of the first sector to erase.
sector_num	unsigned long	The number of sectors to erase.

F_DRIVER

This is the format of the *F_DRIVER* structure:

Element	Type	Description
separated	int	Non-zero if the driver is separated.
user_data	unsigned long	User-defined data.
user_ptr	void *	User-defined pointer.
writesector	F_WRITESECTOR	Write a sector to the drive. This is mandatory if format or any write access is required.
writemultiplesector	F_WRITEMULTIPLESECTOR	Write a series of sectors to the drive. If this is unavailable F_WRITESECTOR may be used.
readsector	F_READSECTOR	Read a sector from the drive.
readmultiplesector	F_READMULTIPLESECTOR	Read a series of sectors from the drive. If this is unavailable F_READSECTOR may be used.
getphy	F_GETPHY	Used to get the physical properties of the drive, such as the number of sectors.
getstatus	F_GETSTATUS	(Only for removable drives) Used to test whether a media has been removed or changed.
release	F_RELEASE	Release any resources associated with a drive when it is freed by the host (file) system.
ioctl	F_IOCTL	Used to send user-defined messages to the driver and get a response.

Media Status Definitions

There are three definitions:

Option	Description
F_ST_MISSING	No media is inserted; the physical socket is empty.
F_ST_CHANGED	The media has been removed and a new (or the same) media has been inserted since the previous status call.
F_ST_WRPROTECT	The media is write-protected.