RX Family

Block Access Media Driver API

R01AN1443EU0100
Rev.1.00
December 20, 2012

Introduction

This document describes a standardized Application programming interface (API) for random-access block oriented storage media device drivers. The API routines described here are intended to be adopted as the interface between high-level application code and the low-level media drivers.

Target Device

The following is a list of devices that have been used to evaluate this API:

- RX62N, RX63N

This API can use other RX devices by changing a startup program.

Contents

1. Overview

Many embedded applications require the storage and retrieval of data or files on memory devices that use block transfer operations as the data transfer method. The high-level operations that access these devices often do so with a common set of methods. It is therefore desirable that different storage media devices be adaptable to various applications without having to rely on interface methods unique to each device. Since the various devices appear logically similar, a common Application programming interface (API) can be used to encapsulate the physical device drivers that are unique to each device.

This document describes a simplified standardized API for random-access block oriented storage media device drivers. The objective is to establish an abstracted interface that provides sufficient flexibility to support a variety of different types of physical storage media hardware. This will permit the storage device to be handled as an object by the higher level communication or application layers, where the lower level device driver will convert the transactions to the required subtype of the target device.

While the language is standard C, the abstracted device drivers can be considered to behave as a 'class'. This permits multiple media devices to coexist in the same system and to be accessed through the same set of function calls.

In addition to the API described, a method of maintaining a collection of device drivers is described. The collection of device drivers is managed as a virtual table, that is, an array of pointers to media driver objects. A media driver object is implemented as a set of pointers to functions that translate from the common API routines to the media specific low-level functions.

2. Organization and design of the media driver

In Figure 1, layer 3 presents the common interface between the higher-level application protocols and the storage media specific driver protocols. The interface functions of the Block Access Media Driver API present a single entry point to the application for the media storage operations. A call to one of the API functions is directed to the desired media device inside the API. The API has access to all installed media drivers by means of an array of pointers to media driver objects. A media driver object is a data structure that contains a set of pointers to functions that carry out the generalized communication tasks.

The driver object's location in the array corresponds to its logical unit number (LUN), which is used by the application to select the target media device for storage operations. When the application is to perform an operation with the media device, it passes the LUN as an argument to the abstracted function call. The LUN serves as an index to the array of driver objects, and the corresponding driver function is then called. Therefore, Layer 3 is an interface layer only. It could be bypassed in systems that will never use more than one specific media device. However, this document only specifies the implementation in which this translation and driver lookup layer is used.

Figure 1, layer 2 consists of the actual implementation of a media driver. It contains a set of functions that comprise the concrete implementation of the layer 3 API interfaces for a given media device. It is in this layer that the API functions are converted to media specific operations in a polymorphic manner. The layer 2 media driver may also contain properties and functions beyond those specified by the API.

At its interface to layer 3, layer 2 converts the arguments passed through calls to the Block Access Media Driver API into the forms required by the underlying media driver. Layer 2 contains protocols specific to handling communication with the media device and may make calls to any other system routines as required.

Figure 1, layer 1 is the set of low-level drivers that control the physical communications channel hardware. Examples of the physical communications channels are, RSPI, RIIC, SCI, GPIO, direct memory access, etc. Layer 1 and layer 2 may be optionally combined; it is not the purpose of this specification to establish the driver design at that level. However, this logical hierarchy represents the typical architecture in which a media driver incorporates one or more additional communications channel drivers or other peripheral drivers below it.

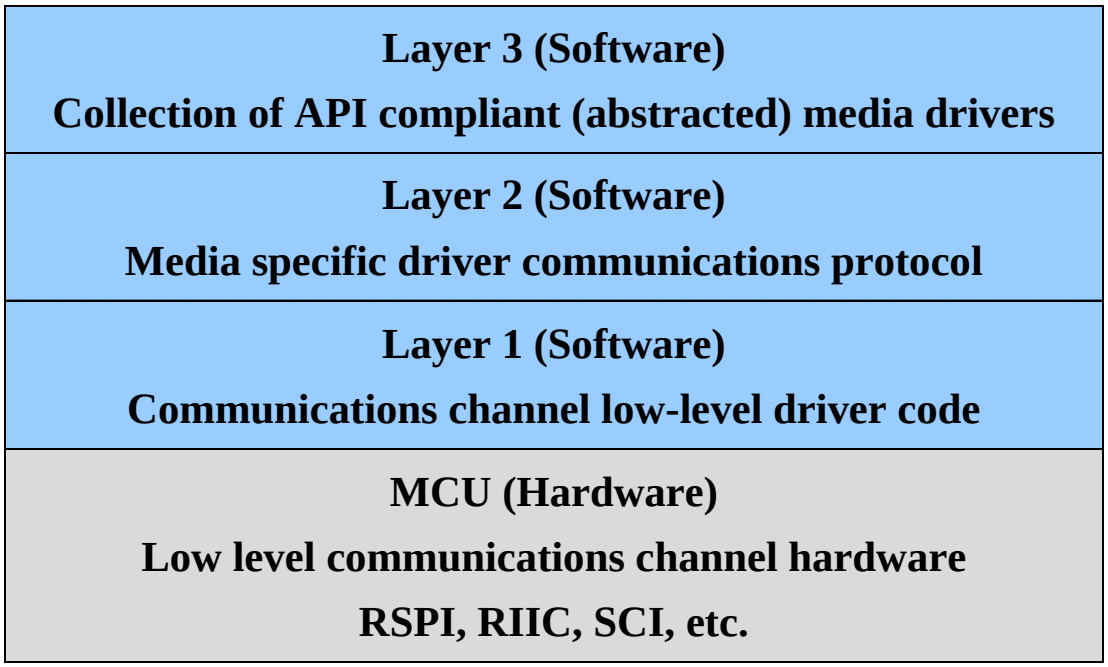

Figure 1 : Relationship of API to Device Drivers

This document focuses on the description of Layer 3 and its boundaries.

2.1 General Characteristics of the Block Access Media Driver

Random-access block oriented storage media devices appear logically similar. This permits a common API to be used to access and control these devices. A basic set of functions can support most of the operations needed to be performed. The basic operations are:

- **Write blocks of data**
  - Identify where the data to be written is located.
    - A reference (pointer) provided by the higher level
  - Identify the location in the media device into which the block of data is to be written.
    - Argument provided by the higher level
    - Specifies the location for the first block
  - Specify how many blocks are to be written
    - Argument provided by the higher level
- **Read blocks of data**
  - Identify the location into which the block of read data is to be copied.
    - A reference (pointer) provided by the higher level
  - Identify the address of the first block in the media device from which to begin reading.
    - Argument provided by the higher level
    - Specifies the location for the first block
  - Specify how many blocks are to be read
    - Argument provided by the higher level
- **Control settings, and query status and configuration**
  - Generic set of commands
    - What is the device's block size?
    - How many blocks does it contain?
    - Query status
    - Initialize/Reset/
    - Close (flush buffers)
    - Retrieve device specific information
    - Perform device specific operations

In addition to the functions that control individual media devices, a number of tasks are required to support and organize multiple block media devices within a single system. Therefore, each device must be able to identify its properties to the system, such as its identity, its capacity, its block size and organization, and its operational status.

- A logical unit number identifies the particular device with which the transaction is to be performed.
- Each device identifies its storage parameters by reporting its block size and total number of block (= total capacity).
- Register the device driver with the system.
- Support resource locking to prevent conflicts from concurrent access of shared resources.

2.2 Supporting Multiple device drivers

References to multiple device drivers are maintained by placing the pointers to the driver objects into a simple array.

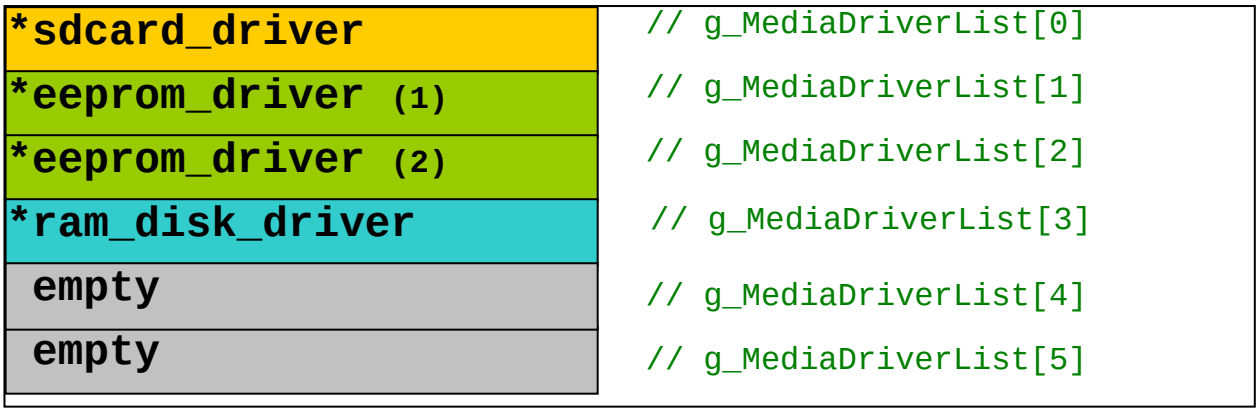

Figure 2 Example of a device driver list

- The LUN parameter serves as the array index to the driver to be used.
- Multiple reference to the same driver allowed:
  - This permits more than one device of the same type to be accessed.
  - The LUN parameter, passed on to the shared driver, informs the driver which of the devices to access.

2.3 Systems using only a single media device

Many systems only require access to a single media storage device. In such cases it is slightly more efficient to bypass the layer of driver abstraction that relies on the g_MediaDriverList[] array of pointers to driver objects. In this case the API can directly call the specific media device driver functions that implement the API call. This will slightly reduce code and memory size, and will improve execution speed a bit by eliminating a couple stages of pointer dereferencing. The API code could be implemented in such a manner as to optionally build for this configuration based on a user configuration setting.

Table 2-1 List of Block Access Media Driver API functions

Function Name	Description
R_MEDIA_Initialize	Registers the media driver
R_MEDIA_Open	Open media driver
R_MEDIA_Open	Close media driver
R_MEDIA_Read	Read from a media device
R_MEDIA_Write	Write to a media device
R_MEDIA_Ioctl	Perform control and query operations on a media device

3. API

3.1 Hardware Requirements

This middleware requires your MCU support the following features:

No MCU dependent requirements.

3.2 Header Files

All API calls are accessed by including a single file *r_media_driver_api.h* which is supplied with this middleware's project code. In addition, the API configuration options must be set by the user in *r_media_driver_api_config.h*

3.3 Integer Types

This project uses ANSI C99 "Exact width integer types" in order to make the code clearer and more portable. These types are defined in *stdint.h*.

3.4 Configuration Overview

This section lists the user configuration requirements to be defined in *r_media_driver_api_config.h* and discusses how they are used.

Table describing configuration options in this middleware	
media_lun_t	This enumeration names the set of media drivers that are present in the system and assigns them numeric values. These constants are used elsewhere by the API as indices to the resources belonging to the corresponding drivers. A final constant macro, MAX_NUM_LUNS, sets the maximum number of logical units that will be supported by the application. It is used in dimensioning the array used to contain the media driver list, and other resources, and is referenced as a limit in supporting code logic. MAX_NUM_LUNS is not set directly; instead it is automatically assigned as part of an enumeration: see section **Media logical unit number enumeration** for an example. It is the listing of target media devices in the enumeration that must be edited by the user to include the media devices that will be used in the system. MAX_NUM_LUNS must always the last item in the enumeration. In this way it will automatically be set to the correct value.
Dependencies	Each media driver must instantiate a global scope structure containing the pointers to its implementations of the media driver API functions. The reference to this structure must be provided to the media driver API by including a reference to it in the *r_media_driver_api_config.h* file. This should normally be found in a header file for the specific media driver, and then that header file would, in-turn, be included in *r_media_driver_api_config.h.*

Table 2 : Info about the configuration

3.5 API Data Structures

This section details the data structures that are used with the middleware's API functions.

3.5.1 Definition of media driver function pointer types

The media driver API interfaces to the specific media device driver functions through an abstraction layer. Therefore each of the API functions has a corresponding type-defined type that consists of a pointer to a function with a matching parameter list.

```
/* Define media driver function pointer types. */

typedef bool       (*media_init_t) (uint8_t lun,
                                    media_driver_t * p_media_driver);

typedef media_ret_t(*media_open_t) (uint8_t lun);

typedef media_ret_t(*media_close_t)(uint8_t lun);

typedef media_ret_t(*media_read_t) (uint8_t lun, uint8_t* p_rbuffer,
                                    uint32_t start_block, uint8_t  block_count);


typedef media_ret_t(*media_write_t)(uint8_t lun, uint8_t* p_wbuffer,
                                    uint32_t start_block, uint8_t  block_count);


typedef media_ret_t(*media_ioctl_t)(uint8_t lun, ioctl_cmd_t ioctl_cmd,
                                    void * ioctl_data);
```

3.5.2 Media driver data structure

This data structure contains the logical unit number of the media storage device and the function pointers to the standardized set of functions that every media driver must implement. To use a media driver, an instance of this structure must be created that has the actual function pointers initialized and assigned to the corresponding structure elements. In this way, the higher level application need only call these generic functions to interface to any media device that implements this interface.

```
/* Media driver Data Structure */

typedef struct media_driver_s
{
    uint8_t            lun;     /* Logical unit number of the storage device. */
    media_open_t       pf_media_open;  /* pointer to driver open function. */
    media_close_t      pf_media_close; /* pointer to driver close function. */
    media_read_t       pf_media_read;  /* pointer to driver read function. */
    media_write_t      pf_media_write; /* pointer to driver write function. */
    media_ioctl_t      pf_media_ctrl;  /* pointer to driver control function. */
} media_driver_t;
```

3.5.3 Media logical unit number enumeration

This enumeration is used to define labels for the logical unit numbers that will be assigned to each media device. The devices defined here are examples. This is intended to be a user configurable list, however media drivers distributed as middleware may predefine certain labels that will need to be used here. The values assigned in this enumeration will determine the location in the drivers list that the driver reference will be placed when the driver is registered. Afterward these values will serve as indices to the list and will be used to dereference calls to the media driver functions via the API. This enumeration must always start at 0 and must be sequential to provide proper array indexing. Use these device labels as the argument for the lun parameter when making calls to the API.

```
/* Examples. User adds devices here */
```

```
typedef enum
{
    SPI_FLASH_LUN = 0,  /* device 0: */
    SDCARD_LUN,         /* device 1: */
    RAM_DISK_LUN        /* device 2: */

    /* Do not add after this line. */
    MAX_NUM_LUNS        /* Do not change this line. */
} media_lun_t;
```

3.5.4 Media driver list

This array is defined to hold a set of pointers to media_driver_t type structures. This provides a means to manage multiple media device drivers in the same system. An individual media driver is accessed by means of indexing into this array by use of the logical unit number (lun) parameter of the media driver API function call. The maximum number of drivers loaded at any given time is limited by the user configurable value MAX_NUM_LUNS.

This list may contain duplicate pointers to the same driver in order to support multiple media devices of the same type.

```
/* Media driver list */

extern media_driver_t * g_MediaDriverList[MAX_NUM_LUNS];
```

3.6 Return Values

This shows the different values API functions can return.

```
/* Return values for functions */
typedef enum
{
   MEDIA_RET_OK = 0,        /* 0: Successful */
   MEDIA_RET_RWERR,         /* 1: Read/Write Error */
   MEDIA_RET_WRPRT,         /* 2: Write Protected */
   MEDIA_RET_NOTRDY,        /* 3: Not Ready */
   MEDIA_RET_PARERR,        /* 4: Invalid Parameter */
   MEDIA_RET_OP_FAIL,       /*  Operation failed. */
   MEDIA_RET_DEV_OPEN       /*  The device is already open. */
   /* For expansion, add only after this line. */
} media_ret_t;
```

3.7 Adding the Media Driver API Middleware to Your Project

The source file *r_media_driver_api.c*, and header files *r_media_driver_api.h*, and *r_media_driver_api_config.h* will need to be added to your project. The code is not hardware specific. You will need to choose whether to build for multiple media driver support or for only a single media driver. Change the settings in the *r_media_driver_api_config.h* file as required for your configuration.

Your project will need one or more block media device drivers that are compliant with the interface requirements of this API. If you use the single media driver build configuration, you will need to edit the code in each of the API functions in *r_media_driver_api.c* to change the placeholder function names to the names of the actual media driver functions that will get called. (TODO: Find a way to do this in the configuration file.).

The Media Driver API code is written to ANSI C99 standard and uses exact width integer types in order to make the code clearer and more portable. These types are defined in *stdint.h*. So your compiler must either support ANSI C99 stdint.h or you will need to create a typedefines file that defines the integer types used by this code.

4. API Functions

4.1 R_MEDIA_Initialize

The R_MEDIA_Initialize() function initializes data structures and variables that are used by the target media device to support its operation for the first time.

Format

```
bool R_MEDIA_Initialize(uint8_t lun, media_driver_t * p_media_driver);
```

Parameters

lun

Logical unit number. Identifies a physical media device or logical partition of a physical media device.

Return Values

TRUE:	*Success*
FALSE:	*Error.*

Properties

Prototyped in file “r_media_driver.h”

Description

The R_MEDIA_Initialize() function will initialize data structures and variables that are used by the target media device to support its operation for the first time. R_MEDIA_Initialize() must be called once before any other operations can be performed on the media device, Among the operations performed by this function is the registration of the media driver to the system loaded drivers list.

Reentrant

- Yes, but only needs to be called once for a given device.

Example

```
/* Prepare the system for use of MMC media driver.  */

if (!R_MEDIA_Initialize(SDCARD_LUN, &g_MmcMediaDriver))
{
    /* Handle the error */
}

/* Media driver resources successfully registered/allocated. */
```

Special Notes:

The initialize function does not actually start operations on the device or initialize device registers. Those tasks will be performed by the R_MEDIA_Open() function.

4.2 R_MEDIA_Open

The R_MEDIA_Open() function initializes the hardware registers for peripherals used by the media driver and leaves the media device ready for communications.

Format

```
media_ret_t    R_MEDIA_Open(uint8_t lun);
```

Parameters

lun

Logical unit number. Identifies a physical media device or logical partition of a physical media device.

Return Values

MEDIA_RET_OK:	*Success*
MEDIA_RET_PARERR:	*Invalid parameter error*
MEDIA_RET_DEV_OPEN:	*The device was already open*
MEDIA_RET_NOTRDY:	*The device is not responding or not present*
MEDIA_RET_OP_FAIL:	*Any other failures*

Properties

Prototyped in file “r_media_driver.h”

Description

The R_MEDIA_Open() function initializes the hardware registers for peripherals used by the media driver and leaves the driver ready for communications. The R_MEDIA_Initialize() must have been called once before this function can be called. R_MEDIA_Open() must only be called once unless the R_MEDIA_Close() function is called. It may be called again to restore a device's settings to their initial state after a device has been closed with the R_MEDIA_Close() function.

Reentrant

- No, but is protected by lock to prevent errors from concurrent function calls.

Example

```
/* Ready the media driver and hardware for communications with the media.  */
result = R_MEDIA_Open(RAM_DISK_LUN);

if (MEDIA_RET_OK != result)
{
    /* Process the error */
}

/* OK to read or write the media now. */
```

4.3 R_MEDIA_Close

The R_MEDIA_Open() function initializes the hardware registers for peripherals used by the media driver and leaves the media device ready for communications.

Format

```
media_ret_t    R_MEDIA_Close(uint8_t lun);
```

Parameters

lun

Logical unit number. Identifies a physical media device or logical partition of a physical media device.

Return Values

MEDIA_RET_OK:	*Success*
MEDIA_RET_PARERR:	*Invalid parameter error*
MEDIA_RET_OP_FAIL:	*Any other failures*

Properties

Prototyped in file “r_media_driver.h”

Description

The R_MEDIA_Close() function flushes any data that may remain in any queues belonging to the media driver, releases all resources previously allocated to the media driver under the R_MEDIA_Initialize() and R_MEDIA_Open() functions, and returns the hardware to an inactive state. This function must only be called for a device that is currently open.

Reentrant

- No, but is protected by lock to prevent errors from concurrent function calls.

Example

```
/* Close the media driver and release its allocated resources.  */
result = R_MEDIA_Close(RAM_DISK_LUN);

if (MEDIA_RET_OK != result)
{
    /* Process the error */
}

/* The device closed successfully, and its resources are no longer in use */
```

4.4 R_MEDIA_Read

The R_MEDIA_Read() function reads one or more blocks of data from the selected media device and places it into a buffer provided by the caller.

Format

```
media_ret_t    R_MEDIA_Read (uint8_t lun,
                             uint8_t* p_rbuffer,
                             uint32_t start_block,
                             uint8_t  block_count);
```

Parameters

lun

Logical unit number. Identifies a physical media device or logical partition of a physical media device.

p_rbuffer

Pointer to destination buffer where driver is to place the read data. Caller must insure that sufficient space is available at the indicated address to hold the requested amount of data.

start_block

The logical block number (or LBA -logical block address) on the media that the media driver should begin reading from. In the case of multi-block reads, this will be the lowest logical block number of the sequence. Block numbering is zero-based.

block_count

The total number of sequential logical blocks to be read, starting at the LBA indicated by the start_block parameter.

Return Values

MEDIA_RET_OK:	*Success*
MEDIA_RET_PARERR:	*Invalid parameter error*
MEDIA_RET_RWERR:	*Read/Write Error*
MEDIA_RET_NOTRDY:	*Not Ready*
MEDIA_RET_OP_FAIL:	*Any other failures*

Properties

Prototyped in file “r_media_driver.h”

Description

The R_MEDIA_Read() function reads one or more blocks of data from the media device and places it into a buffer provided by the caller. This function must only be called for a device that is currently open.

Reentrant

- No, but is protected by lock to prevent errors from concurrent function calls.

Example

```
/* Read 1 block of data from the RAM-disk starting at block 2. */
lba = 2;

result = R_MEDIA_Read(RAM_DISK_LUN, &buffer, lba, 1);

if (MEDIA_RET_OK != result)
{
    /* Process the error */
}

/* The data was read successfully and is now present in the read buffer. */
```

4.5 R_MEDIA_Write

The R_MEDIA_Write() function writes one or more blocks of data to the selected media device from a source buffer provided by the caller .

Format

```
media_ret_t    R_MEDIA_Write (uint8_t lun,
                              uint8_t* p_wbuffer,
                              uint32_t start_block,
                              uint8_t  block_count);
```

Parameters

lun

Logical unit number. Identifies a physical media device or logical partition of a physical media device.

p_wbuffer

Pointer to source buffer that contains the data that is to be written to the media device.

start_block

The logical block number (or LBA -logical block address) on the media at which the media driver should start writing. In the case of multi-block writes, this will be the lowest logical block number of the sequence. Block numbering is zero-based.

block_count

The total number of sequential logical blocks to be written, starting at the LBA indicated by the start_block parameter.

Return Values

MEDIA_RET_OK:	*Success*
MEDIA_RET_PARERR:	*Invalid parameter error*
MEDIA_RET_RWERR:	*Read/Write Error*
MEDIA_RET_WRPRT:	*Write Protected*
MEDIA_RET_NOTRDY:	*Not Ready*
MEDIA_RET_OP_FAIL:	*Any other failures*

Properties

Prototyped in file “r_media_driver.h”

Description

The R_MEDIA_Write() function writes one or more blocks of data from the location pointed to by p_wbuffer to the media device. This function must only be called for a device that is currently open.

Reentrant

- No, but is protected by lock to prevent errors from concurrent function calls.

Example

```
/* Write 1 block of data to the RAM-disk starting at block 0. */
lba = 0;

result = R_MEDIA_Write(RAM_DISK_LUN, &buffer, lba, 1);

if (MEDIA_RET_OK != result)
{
    /* Process the error */
}

/* The data was written to the media device successfully. */
```

4.6 R_MEDIA_Ioctl

The R_MEDIA_Ioctl() function provides a generalized means to pass special command and control instructions to the media driver, and for the driver to return information.

Format

```
media_ret_t    R_MEDIA_Ioctl (uint8_t lun,
                              ioctl_cmd_t ioctl_cmd,
                              void * ioctl_data);
```

Parameters

lun

Logical unit number. Identifies a physical media device or logical partition of a physical media device.

ioctl_cmd

Enumerated type that defines the particular command that the driver is to execute. The ioctl command enumeration list, ioctl_cmd_t, contains a predefined list of commands that is referenced by all API compliant drivers.

ioctl_data

This is a pointer to an area of memory that can contain additional data required for the driver to complete the ioctl command, and/or return additional information to the caller. The content of this data is predefined for certain predefined ioctl commands, but it is undefined for custom ioctl commands.

Return Values

MEDIA_RET_OK:	*Success*
MEDIA_RET_PARERR:	*Invalid parameter error*
MEDIA_RET_NOTRDY:	*Not Ready*
MEDIA_RET_OP_FAIL:	*Any other failures*

Properties

Prototyped in file “r_media_driver.h”

Description

The R_MEDIA_Ioctl() function provides a generalized means to pass special command and control instructions to the media driver, and for the driver to return information. The ioctl command enumeration list, ioctl_cmd_t, contains a predefined list of commands that is referenced by all API compliant drivers. Additional information that may be needed by the driver to process the command can be stored at the memory location pointed to by the ioctl_data parameter. In general, the format of data stored at ioctl_data is variable and must be defined for each driver. However, there is a limited set of predefined ioctl commands to which every driver must be capable of responding.

Reentrant

- No, but is protected by lock to prevent errors from concurrent function calls.

Example

```
/* Query the driver for unit 0 to get the number and size of the data blocks
contained in the media device. */
lun = 0;
uint32_t num_blocks;
uint32_t block_size;
uint64_t capacity; // for media capacity larger than 4GB.

result = R_MEDIA_Ioctl(lun, MEDIA_GET_BLOCK_SIZE,  &block_size);
result = R_MEDIA_Ioctl(lun, MEDIA_GET_BLOCK_COUNT, &num_blocks);

/* block_size and num_blocks now contain the reported values. */
capacity = (uin64_t)block_size * (uint64_t)num_blocks;
```

.

Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry

Revision Record

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Dec.30.12		First version

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accord with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
3. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; and safety equipment etc.
   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (nuclear reactor control systems, military equipment etc.). You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application for which it is not intended. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You should not use Renesas Electronics products or technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. When exporting the Renesas Electronics products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
11. This document may not be reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700, Fax: +44-1628-651-804

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 2.2