RE01 1500KB Group

R01AN4945EJ0100
Rev.1.00
Sep 30, 2019

USB Peripheral Mass Storage Class Driver (PMSC) Specification

Introduction

This documentation describes the specification of USB Peripheral Mass Storage Class of CMSIS software package..

Target Device

RE01 1500KB Group

Contents

1. Overview

This driver, when used in combination with USB Basic Peripheral Driver, operates as a USB peripheral mass storage class driver (PMSC). The USB peripheral mass storage class driver (PMSC) comprises a USB mass storage class bulk-only transport (BOT) protocol. When combined with a USB peripheral control driver and media driver, it enables communication with a USB host as a BOT-compatible storage device.

This driver supports the following functions.

- Storage command control using the BOT protocol
- Response to mass storage device class requests from a USB host

1.1 Please be sure to read

Please refer to the document (Document number: R01AN4944) for *USB Basic Peripheral Driver Specification* when creating an application program using this driver.

1.2 Limitation

1. This driver returns the value 0 (zero) to the mass storage command (*GetMaxLun*) sent from USB Host.
2. The sector size which this driver supports is 512 only.

1.3 Note

1. This driver is not guaranteed to provide USB communication operation.
2. The user needs to implement the media driver function which controls the media area used as the storage area.

1.4 Terms and Abbreviations

Terms and abbreviations used in this document are listed below.

APL	:	Application program
BOT	:	Bulk Only Transport.
DDI	:	Device Driver Interface, or PMSDD API.
IDE	:	Integrated Development Environment
Non-OS	:	USB Driver for OS-less
PCD	:	Peripheral Control Driver for USB-BASIC-FW
PCI	:	PCD Interface
PMSCD	:	Peripheral Mass Storage Class Driver (PMSCF + PCI + DDI)
PMSCF	:	Peripheral Mass Storage Class Function
PMSDD	:	Peripheral Mass Storage Device Driver (ATAPI driver)

2. Software Configuration

This driver comprises two layers: PMSCD and PMSDD.

PMSCD comprises three layers: PCD API (PCI), PMSDD API (DDI), and BOT protocol control and data sends and receives (PMSCF).

PMSCD uses the BOT protocol to communicate with the host via PCD.

PMSDD analyzes and executes storage commands received from PMSCD. PMSDD accesses media data via the media driver.

Figure 2-1 shows the configuration of the modules.

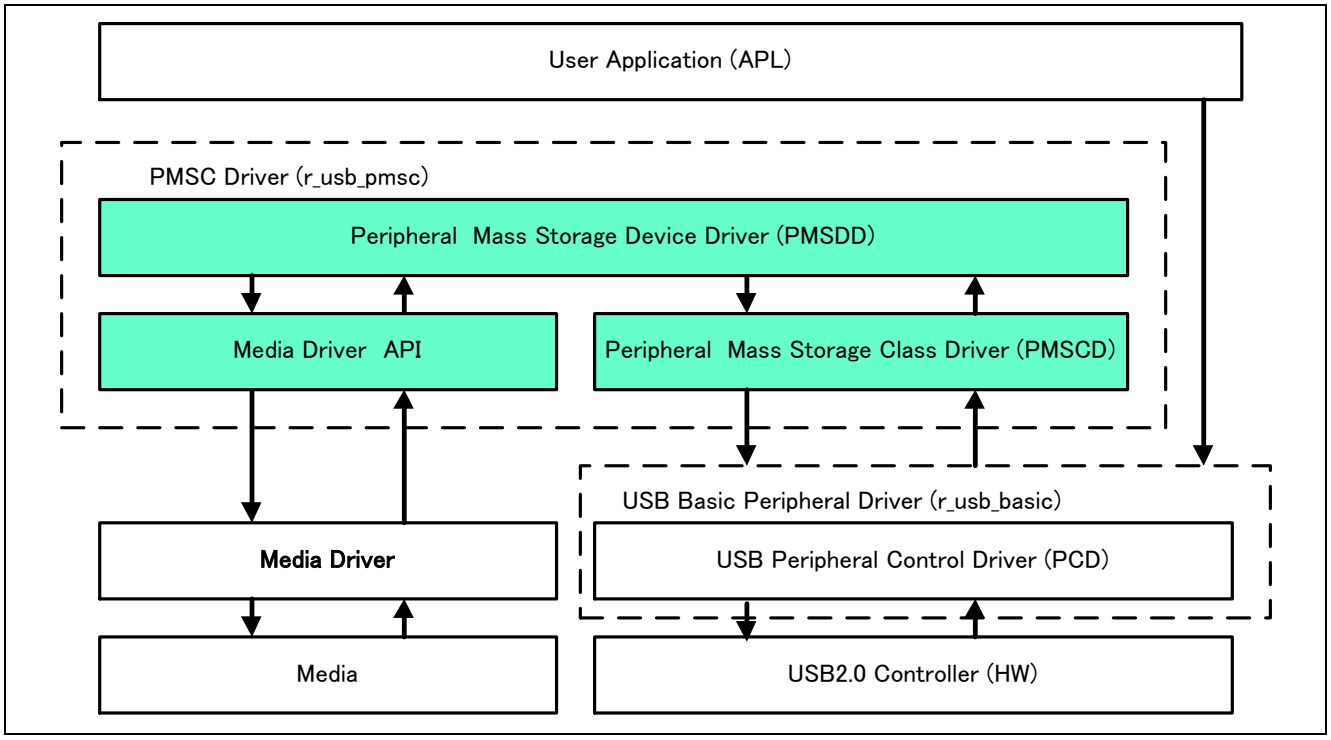

Figure 2-1 Software Structure

Table 2-1 Module Function Overview

Module	Description
PMSDD	Mass Storage Device Driver ・ Processes storage commands from the PMSCD ・ Accesses media via the media driver
DDI	PMSDD-PMSCD interface function
PMSCF	Mass Storage Class Driver ・ Controls BOT protocol data and responds to class requests. ・ Analyzes CBWs and transmits/receives data. ・ Generates CSWs together with the PMSDD/PCD.
PCI	PMSCD – PCD interface function
PCD	USB Peripheral H/W Control driver

3. Class Driver Overview

3.1 Class Requests

Table 3-1 lists the class requests supported by this driver

Table 3-1 MSC Class Requests

Request	Code	Description
Bulk-Only Mass Storage Reset	0xFF	Resets the connection interface to the mass storage device.
Get Max Lun	0xFE	Reports the logical numbers supported by the device.

3.2 Storage Commands

Table 3-2 lists the storage commands supported by this driver. This driver send the STALL or FAIL error (CSW) to USB HOST when receiving other than the following command.

Table 3-2 Storage Commands

Command	Code	Description
TEST_UNIT_READY	0x00	Checks the state of the peripheral device.
REQUEST_SENSE	0x03	Gets the error information of the previous storage command execution result.
INQUIRY	0x12	Gets the parameter information of the logical unit.
READ_FORMAT_CAPACITY	0x23	Gets the formattable capacity.
READ_CAPACITY	0x25	Gets the capacity information of the logical unit.
READ10	0x28	Reads data.
WRITE10	0x2A	Writes data.
MODE_SENSE10	0x5A	Gets the parameters of the logical unit.

4. Peripheral Device Class Driver (PDCD)

4.1 Basic Functions

The functions of PDCD are to:

1. Supporting SFF-8070i (ATAPI)
2. Respond to mass storage class requests from USB host.

4.2 BOT Protocol Overview

BOT (USB MSC Bulk-Only Transport) is a transfer protocol that, encapsulates command, data, and status (results of commands) using only two endpoints (one bulk in and one bulk out).

The ATAPI storage commands and the response status are embedded in a “Command Block Wrapper” (CBW) and a “Command Status Wrapper” (CSW).

Figure 4-1 shows an overview of how the BOT protocol progresses with command and status data flowing between USB host and peripheral.

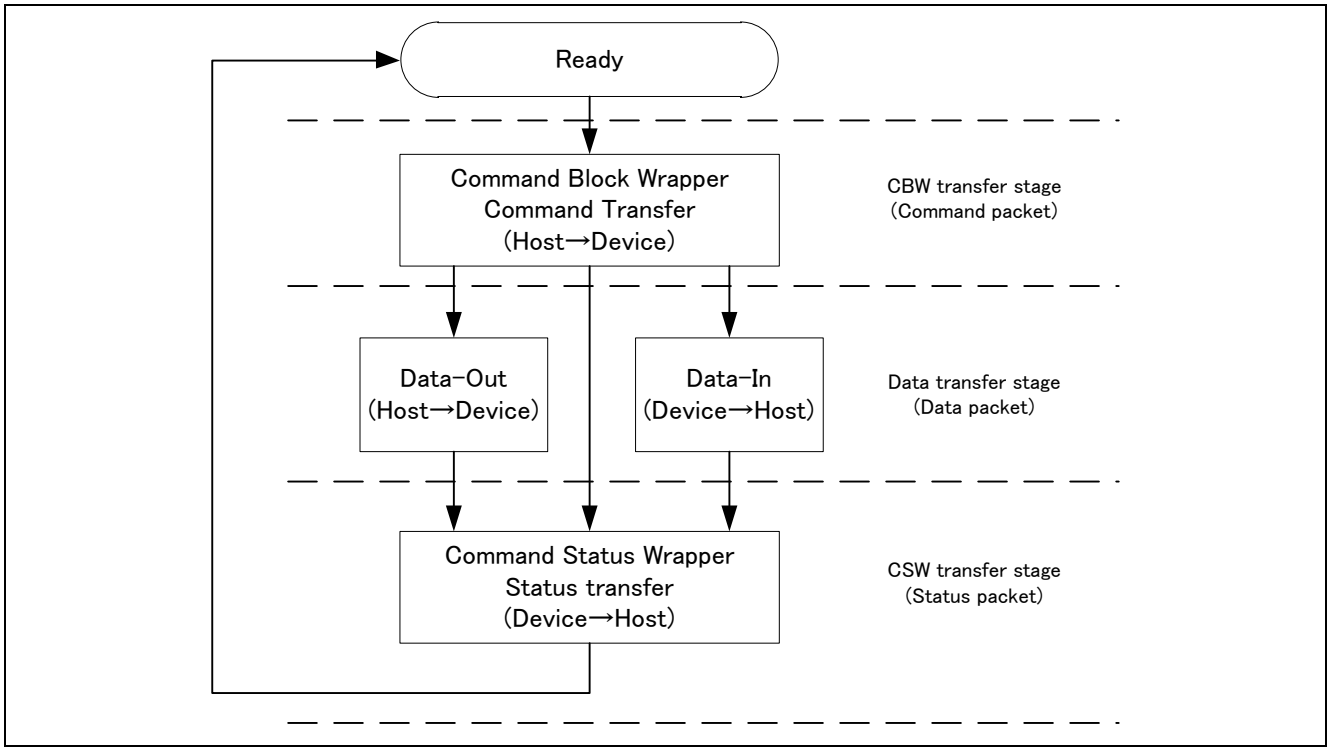

Figure 4-1 BOT protocol Overview.
Command and status flow between USB host and peripheral.

5. API Functions

For API used in the application program, refer to chapter "**API Functions**" in the document (Document number: R01AN4944) for *USB Basic Peripheral Driver Specification*.

6. Configuration (r_usb_pmsc_mini_cfg.h)

Please set the following according to your system.

Note:

Be sure to set *r_usb_basic_mini_cfg.h* file as well. For *r_usb_basic_mini_cfg.h* file, refer to chapter "**Configuration**" in the document (Document number: R01AN4944) for *USB Basic Peripheral Driver Specification*.

1. Setting the response data for Inquiry command.

This driver sends the data specified in the following definitions to the USB Host as the response data of Inquiry command.

(1). Setting Vendor Information

Specify the vendor information which is response data of Inquiry command. Be sure to enclose data of 8 bytes with double quotation marks.

```
#define    USB_CFG_PMSC_VENDOR          Vendor Information
```

e.g)

```
#define    USB_CFG_PMSC_VENDOR          "Renesas "
```

(2). Setting Product Information

Specify the product information which is response data of Inquiry command. Be sure to enclose data of 16 bytes with double quotation marks.

```
#define    USB_CFG_PMSC_PRODUCT         Product Information
```

e.g)

```
#define    USB_CFG_PMSC_PRODUCT         "Mass Storage    "
```

(3). Setting Product Revision Level

Specify the product revision level which is response data of Inquiry command. Be sure to enclose data of 4 bytes with double quotation marks.

```
#define    USB_CFG_PMSC_REVISION        Product Revision Level
```

e.g)

```
#define    USB_CFG_PMSC_REVISION        "1.00"
```

2. Setting the number of transfer sector

Specify the maximum sector size to request to PCD (Peripheral Control Driver) at one data transfer. This driver specifies the value of "1 sector (512) × USB_CFG_PMSC_TRANS_COUNT" bytes to PCD as the transfer size. By increasing this value, the number of data transfer requests to the PCD decreases, so the transfer speed performance may be improved. However, note that "1 sector (512) × USB_CFG_PMSC_TRANS_COUNT" bytes of RAM will be consumed.

```
#define    USB_CFG_PMSC_TRANS_COUNT     Number of transfer sectors (1 to 255)
```

e.g)

```
#define    USB_CFG_PMSC_TRANS_COUNT     4
```

7. Media Driver Interface

PMSC uses a common media driver API function to access to the media drivers with different specifications.

7.1 Overview of Media Driver API Functions

Media driver API functions are called by the PMSC and the API functions call the media driver function implemented by the user. This chapter explains the prototype of the media driver API function and the processing necessary for implementing each function.

Table 7-1 shows the list of the media driver API functions.

Table 7-1 Media Driver API

Media Driver API	Processing Description
R_USB_media_initialize	Initializes the media driver.
R_USB_media_open	Opens the media driver.
R_USB_media_close	Closes the media driver.
R_USB_media_read	Reads from the media.
R_USB_media_write	Writes to the media.
R_USB_media_ioctl	Processing the control instructions specific to the media device.

7.1.1 R_USB_media_initialize

Register the media driver function to the media driver

Format

bool R_USB_media_initialize(media_driver_t * p_media_driver);

Arguments

p_meida_driver	Point to the structure area for the media driver

Return Value

TRUE	Successfully completed
FALSE	Error generated

Description

This API registers the media driver function implemented by the user to the media driver.

Be sure to call this API at the initialization processing etc in the user application program.

Note

1. The user needs to implement the media driver function based on the contents described in the above "Arguments", "Return Value" and "Description" etc.
2. For how to register of the media driver function implemented by the user, refer to the chapter **7.3, Registration of the storage media driver**.
3. This API does not do the media device initialization processing and does not do the starting operation processing of the media device. These processing is done by *R_USB_media_open* function.
4. PMSC does not support the function to register the multiple type media driver function.

Example

```
if (!R_USB_media_initialize(&g_ram_mediadriver))
{
    /* Handle the error */
}
result = R_USB_media_open();
if (USB_MEDIA_RET_OK != result)
{
    /* Process the error */
}
```

7.1.2 R_USB_media_open

Initialize the media driver and the media device

Format

usb_media_ret_t R_USB_media_open(void);

Arguments

--

Return Value

USB_MEDIA_RET_OK	Successfully completed
USB_MEDIA_RET_PARAERR	Parameter error
USB_MEDIA_RET_DEV_OPEN	The device was already opened
USB_MEDIA_RET_NOTRDY	The device is not responding or not present
USB_MEDIA_RET_OP_FAIL	Any other failure

Description

This API initializes the media device and the media driver and make the media device and the media driver the ready status.

Be sure to call this API at the initialization processing etc in the user application program.

Note

1. *R_USB_media_initialize* function has to be called before calling this API.
2. The number of calls this API is only once unless *R_USB_media_close* is called. After calling *R_USB_media_close* function, this API can be called again to return the device to the initial state.
3. The user needs to implement the media driver function based on the contents described in the above "Arguments", "Return Value" and "Description" etc.

Example

```
if (!R_USB_media_initialize(&g_ram_mediadriver))
{
    /* Handle the error */
}

result = R_USB_media_open();
if (USB_MEDIA_RET_OK != result)
{
    /* Process the error */
}
```

7.1.3 R_USB_media_close

Release the resource for the media driver and return the media device to the non active state.

Format

usb_media_ret_t R_USB_media_close(void);

Arguments

--

Return Value

USB_MEDIA_RET_OK	Successfully completed
USB_MEDIA_RET_PARAERR	Parameter error
USB_MEDIA_RET_OP_FAIL	Any other failure

Description

This API releases the resource for the media driver and return the media device to the non active state.

Note

1. *R_USB_media_initialize* function has to be called before calling this API.
2. The user needs to implement the media driver function based on the contents described in the above "Arguments", "Return Value" and "Description" etc.

Example

```
result = R_USB_media_close();
if (USB_MEDIA_RET_OK != result)
{
   /* Process the error */
}
```

7.1.4 R_USB_media_read

Read the data blocks from the media device

Format

usb_media_ret_t R_USB_media_read(uint8_t *p_buf, uint32_t lba, uint8_t count);

Argument

p_buf	Pointer to the area to store the read data from the media device
lba	Read start logical block address
count	Number of read block (Number of sector)

Return Value

USB_MEDIA_RET_OK	Successfully completed
USB_MEDIA_RET_PARAERR	Parameter error
USB_MEDIA_RET_NOTRDY	The device is not ready state
USB_MEDIA_RET_OP_FAIL	Any other failure

Description

This API reads the data blocks from the media device. (Read the data blocks for the number of blocks specified by the third argument (*count*) from the LBA (Logical Block Address) specified by the second argument.)

The read data is stored in the specified area by the first argument (*p_buf*).

Note

1. *R_USB_media_initialize* function has to be called before calling this API.
2. The user needs to implement the media driver function based on the contents described in the above "Arguments", "Return Value" and "Description" etc.

Example

```
result = R_USB_media_read(&buffer, lba, 1);
if (USB_MEDIA_RET_OK != result)
{
   /* Process the error */
}
```

7.1.5 R_USB_media_write

Write the data block to the media device

Format

usb_media_ret_t R_USB_media_write(uint8_t *p_buf, uint32_t lba, uint8_t count);

Arguments

p_buf	Pointer to the area where data to be written to the media device is stored
lba	Write start logical block address
count	Number of write blocks (Number of sector)

Return Value

USB_MEDIA_RET_OK	Successfully completed
USB_MEDIA_RET_PARAERR	Parameter error
USB_MEDIA_RET_NOTRDY	The device is not ready state
USB_MEDIA_RET_OP_FAIL	Any other failure

Description

This API write the data blocks to the media device. (Write the data blocks for the number of blocks specified by the third argument (*count*) to the LBA (Logical Block Address) specified by the second argument.)

Store the write data in the area specified by the first argument (*p_buf*).

Note

1. *R_USB_media_initialize* function has to be called before calling this API.
2. The user needs to implement the media driver function based on the contents described in the above "Arguments", "Return Value" and "Description" etc.

Example

```
result = R_USB_media_write(&buffer, lba, 1);
if (MEDIA_RET_OK != result)
{
/* Process the error */
}
```

7.1.6 R_USB_media_ioctl

Get the information of the media driver etc

Format

usb_media_ret_t R_USB_media_ioctl(ioctl_cmd_t command, void *p_data);

Arguments

command	Command code
p_data	Pointer to the area to store the media information

Return Value

USB_MEDIA_RET_OK	Successfully completed
USB_MEDIA_RET_PARAERR	Parameter error
USB_MEDIA_RET_NOTRDY	The device is not ready state
USB_MEDIA_RET_OP_FAIL	Any other failure

Description

This API gets the return information from the media driver by specifying the media driver specific command.

PMSC uses the following commands as the command code to the media driver.

MEDIA_IOCTL_GET_NUM_BLOCKS	Number of block for the media area
MEDIA_IOCTL_GET_BLOCK_SIZE	1 block size

Note

1. *R_USB_media_initialize* function has to be called before calling this API.
2. The user can ndefine the command code specified in the argument(command) newly.
3. The user needs to implement the media driver function based on the contents described in the above "Arguments", "Return Value" and "Description" etc.

Example

```
uint32_t num_blocks;
uint32_t block_size;
uint64_t capacity;.

result = R_USB_media_ioctl(MEDIA_IOCTL_GET_NUM_BLOCKS, (void *)&num_blocks);
result = R_USB_media_ioctl(MEDIA_IOCTL_GET_BLOCK_SIZE, (void *)&block_size);

capacity = (uin64_t)block_size * (uint64_t)num_blocks;
```

7.2 Structure / Enum type definition

The following shows the structure and enum type used by the media driver API.

These are defined in *r_usb_media_driver_if.h* file.

7.2.1 usb_media_driver_t (Structure)

usb_media_driver_t is the structure to hold the pointer to the media driver function implemented by the user.

The following shows *usb_media_driver_t* structure.

```
typedef struct media_driver_t
{
        usb_media_open_t    pf_media_open;    /* Pointer to the open function */
        usb_media_close_t   pf_media_close;   /* Pointer to the close function  */
        usb_media_read_t    pf_media_read;    /* Pointer to the read function */
        usb_media_write_t   pf_media_write;   /* Pointer to the write function */
        usb_media_ioctl_t   pf_media_ctrl;    /* Pointer to the control function */
} usb_media_driver_t
```

7.2.2 usb_media_ret_t (Enum)

The return value is defined in *usb_media_ret_t* (Enum).

```
typedef enum
{
        USB_MEDIA_RET_OK = 0,          /* Successfully Completed */
        USB_MEDIA_RET_NOTRDY,          /* The device is not ready state */
        USB_MEDIA_RET_PARERR,          /* Parameter error */
        USB_MEDIA_RET_OP_FAIL,         /* Any other failure */
        USB_MEDIA_RET_DEV_OPEN,        /* The device was already opened */
} usb_media_ret_t
```

7.2.3 ioctrl_cmd_t (Enum)

The command code specified in the argument of the *R_USB_media_ioctl* function is defined in *ioctl_cmd_t* (Enum).

```
typedef enum
{
    USB_MEDIA_IOCTL_GET_NUM_BLOCKS,    /* Get the number of the logical block */
    USB_MEDIA_IOCTL_GET_BLOCK_SIZE,    /* Get the logical block size */
} ioctl_cmd_t
```

Note:

Please add the command code in the *ioctl_cmd_t* when adding the user own command code.

7.3 Registration of the storage media driver

To change the PMSC's storage media from RAM to something else, such as flash memory, the user has to implement media driver functions to handle reading from and writing to the new storage media and register them to the media driver API functions.

The example below shows the procedure for changing from RAM media to serial SPI flash.

1. Creating Media Driver Functions

Assume that the following functions are implemented by the user as media driver functions for serial SPI flash.

1. usb_media_ret_t spi_flash_open (void)
2. usb_media_ret_t spi_flash_close (void)
3. usb_media_ret_t spi_flash_read(uint8_t *p_buf,uint32_t lba, uint8_t count)
4. usb_media_ret_t spi_flash_write(uint8_t *p_buf,uint32_t lba, uint8_t count)
5. usb_media_ret_t spi_flash_ioctl(ioctl_cmd_t ioctl_cmd,void * ioctl_data)

2. Registering the Media Driver Functions with the Media API

(1). Define the structure *usb_media_driver_t* for the serial SPI flash. As the members of this structure, specify pointers to the relevant media driver functions.

```
struct media_driver_t  g_spi_flash_mediadriver =
{
   &spi_flash_open,
   &spi_flash_close,
   &spi_flash_read,
   &spi_flash_write,
   &spi_flash_ioctl
};
```

(2). In the application program, specify the pointer to *usb_media_driver_t* structure to the argument in *R_USB_media_initialize* function (API), and perform initialization processing.

```
== Application Program ==

R_USB_media_initialize(& g_spi_flash_mediadriver );
```

The serial SPI flash function is registered as the media driver function called by the media drvier by doing the above order.

7.4 Implementation of the strorage media dirver

The user needs to implement the media driver function for controlling the storage media to be used.

The implemented media driver function is called from PMSC via the API described in chapter 7.1, **Overview of Media Driver API Functions** from PMSC.

Note:

For the necessary processing to implement the media driver function, refer to each API specification described in chapter **7.1, Overview of Media Driver API Functions.**

7.5 Prototype Declaration of Media Driver function

The following shows the prototype declaration of the media driver function.

```
1. usb_media_ret_t  (*media_open_t) (uint8_t);                                  /* Open function type */
2. usb_media_ret_t  (*media_close_t)(uint8_t);                                  /* Close function type */
3. usb_media_ret_t (*media_read_t)(uint8_t, uint8_t*, uint32_t, uint8_t);       /* Read function type */
4. usb_media_ret_t (*media_write_t)(uint8_t, uint8_t*, uint32_t, uint8_t);      /* Write function type */
5. usb_media_ret_t  (*media_ioctl_t)(uint8_t, ioctl_cmd_t, void *);             /* Control function type */
```

8. Creating an Application

Refer to the chapter "**Creating an Application Program**" in the document (Document number: R01AN4944) for *USB Basic Peripheral Driver Specification*.

Note:

Be sure to call *R_USB_media_initialize* function (API) and *R_USB_media_open* function (API) at the initialize processing etc in the user application program.

Website and Support

Renesas Electronics Website

http://www.renesas.com/

Inquiries

http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Sep 30, 2019	—	First edition issued

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit 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 accordance 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 Microprocessing unit or Microcontroller unit products 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.