Renesas USB MCU

R01AN0710EJ0215
Rev.2.15
Mar 28, 2016

USB Peripheral Mass Storage Class Driver (PMSC) using Basic Mini Firmware

Introduction

This document is an application note for the USB Peripheral Mass Storage Class Driver (PMSC) built using the USB Basic Mini Firmware.

Target Device

RL78/G1C, RL78/L1C, R8C/3MU, R8C/34U, R8C/3MK, R8C/34K

This program can be used with other microcontrollers that have the same USB module as the above target devices. When using this code in an end product or other application, its operation must be tested and evaluated thoroughly.

This program has been evaluated using the corresponding MCU's Renesas Starter Kit board.

Contents

1. Overview

This document is a manual describing use of the USB Peripheral Mass Storage Class Driver (PMSC) for Renesas USB MCU.

1.1 Functions and Features

The USB Peripheral Mass Storage class driver comprises a USB Mass Storage class bulk-only transport (BOT) protocol. When combined with a USB peripheral control driver and storage device driver, it enables communication with a USB host as a BOT-compatible storage device.

The PMSC software cannot itself read/write the storage media. See 11.

1.2 Related Documents

1. Universal Serial Bus Revision 2.0 specification
2. USB Mass Storage Class Specification Overview Revision 1.1
3. USB Mass Storage Class Bulk-Only Transport Revision 1.0, "BOT" protocol
   [http://www.usb.org/developers/docs/]
4. User's Manual: Hardware
5. USB-BASIC-F/W Application Note
6. Block Access Media Driver API.
   Available from Renesas Electronics Website

Renesas Electronics Website
http:// www.renesas.com/
USB Devices Page
http://www.renesas.com/prod/usb/

1.3 Terms and Abbreviations

API	:	Application Program Interface
APL	:	Application program
BOT	:	Universal Serial Bus Mass Storage Class Bulk-Only Transport (Available at USB Implementers Forum)
cstd	:	Prefix of Function and File for Host & Peripheral USB-BASIC-FW
DDI	:	Device driver interface, or PMSDD API.
H/W	:	Renesas USB MCU
PCD	:	Peripheral control driver of USB-BASIC-FW
PDCD	:	Peripheral device class driver (device driver and USB class driver)
PCI	:	PCD interface
PMSCD	:	Peripheral mass storage USB class driver (PMSCF + PCI + DDI)
PMSCF	:	Peripheral mass storage class function
PMSDD	:	Peripheral mass storage device driver (sample ATAPI driver)
PP	:	Pre-processed definition
pstd	:	Prefix of Function and File for Peripheral USB-Basic-F/W
RSK	:	Renesas Starter Kits
Scheduler	:	Used to schedule functions, like a simplified OS.
Scheduler Macro	:	Used to call a scheduler function (non-OS)
SW1/SW2/SW3	:	User switches on the RSK Board
Task	:	Processing unit
USB	:	Universal Serial Bus
USB-BASIC-FW	:	USB Basic Firmware mini for Renesas USB device (non-OS)

2. Operating Confirmation Environment

2.1 Compiler

The compilers which is used for the operating confirmation are follows.

a. CA78K0R Compiler V.1.71

b. CC-RL Compiler V.1.01

c. IAR C/C++ Compiler for RL78 version 2.10.4

d. KPIT GNURL78-ELF v15.02

e. C/C++ Compiler Package for M16C Series and R8C Family V.6.00 Release 00

2.2 Evaluation Board

The evaluation boards which is used for the operating confirmation are follows.

a. Renesas Starter Kit for RL78/G1C (Product No: R0K5010JGC001BR)

b. Renesas Starter Kit for RL78/L1C (Product No: R0K50110PC010BR)

c. R8C/34K Group USB Peripheral Evaluation Board (Product No: R0K5R8C34DK2PBR)

3. Software Configuration

3.1 Module Configuration

As shown in Figure 3-1, PDCD comprises two layers: The class driver PMSCD and the device driver PMSDD.

PMSCD comprises three layers: PCD API (PCI) closest to the USB HW layer, on top of that the PMSDD API (DDI), with the BOT protocol control and data transmission/reception (PMSCF) layer in-between.

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 3-1 shows a block diagram of the SW modules.

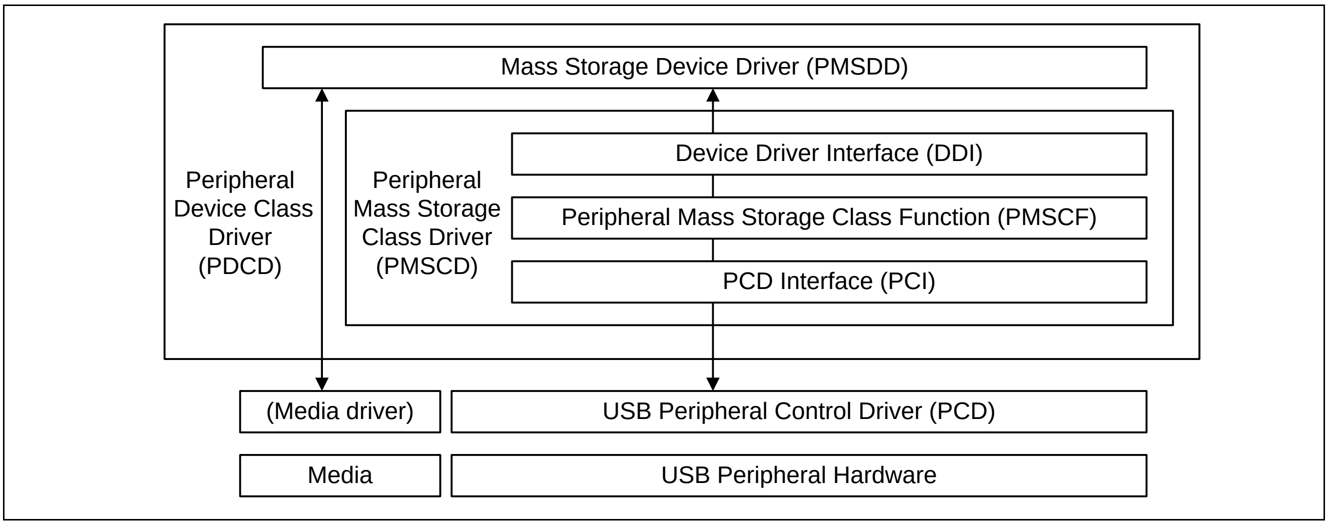

Figure 3-1 Software Block Diagram

3.1.1 PDCD

As shown in Figure 3-1, PDCD incorporates PMSDD and PMSCD. It takes care of class requests from the USB host, and responds to USB host storage commands.
Table 3-1 provides an overview of the parts of PDCD, as well as PCD and the media driver. The media driver is implemented as an interchangeable block media type storage driver.

3.1.2 PMSCD

PMSCD itself in turn comprises three layers: PMSCF, which performs BOT protocol control and data transmission/reception; DDI for interfacing with PMSDD; and a group of functions (PCI) for interfacing with PCD. The main functions of these layers are as follows.

1. PMSCF:

   USB mass storage class BOT protocol control

   CBW analysis, data transmission/reception, and CSW creation in coordination with PMSDD/PCD

   Responding to class requests (MassStorageReset, GetMaxLUN)

2. PCI :

   Processing of tasks, message boxes, and memory pools during configuration and detach

   Receiving class requests

   Clearing STALL states and setting related callback functions

   Setting structures and callback functions for PCD transmit/receive data

3. DDI:

   Driver registration.

   Data transfer information and execution results of PMSDD execution. PMSCD is notified via the ATAPI command result callback function .

Table 3-1 Overview of Modules

Chapter this doc	Module	Description	Reference folder/file	Note
USB Basic FW	PCD	USB peripheral hardware control driver.	src/USBSTDFW	See "Renesas USB MCU USB Basic Firmware mini Application note
Ch. 6	PCI	PMSCF-PCD interface functions.	src/MSCFW/PMSC/r_usb_pmsc_pci.c	
	PMSCF	Core component of PMSCD. Controls BOT protocol data and responds to USB class requests. Also transfers storage commands and data to and from storage (PMSDD).	src/MSCFW/PMSC/ r_usb_pmsc_request.c r_usb_pmsc_driver.c	
	DDI	PMSDD-PMSCF interface : Driver registration and ATAPI result callback.	src/MSCFW/PMSC/r_usb_pmsc_ddi.c	
Ch. 7	PMSDD	Peripheral mass storage media driver. It processes storage commands from PMSCD and accesses the media via the block media driver below. (To be modified to match the memory device.)	src/MSCFW/MEDIA/r_usb_atapi_driver.c	
Ch. 8	Block Media Driver	Block media storage driver.	src/MSCFW/MEDIA/r_usb_atapi_memory.c	See Block Media API application note

3.2 File Structure

The following shows the folder structure for the files provided in this device class.

The source codes unique to each device and evaluation board are stored in the corresponding hardware resource folder (HwResource)*.

```
workspace
  ＋ [ RL78 / R8C ]
    ＋[ CCRL / CS+ / IAR / e2 studio / HEW ]
      ＋[RL78G1C / RL78L1C / R8C3MK / R8C3MU / R8C34K / R8C34U ]
          ＋ PERI                                  Build result
          ＋ src
            ＋──── media_driver [Media driver for Mass Storage Class ]
            │        ＋──── eeprom                 EEPROM driver
            ＋──── MSCFW [Mass Storage Class driver ]
            │        ＋──── inc                    Common header file of MSC driver
            │        ＋──── MEDIA                  Media driver
            │        ＋──── PMSC                   MSC driver
            ＋────SmplMain [ Sample Application ]
            │        ＋──── APL                    Sample application
            ＋────USBSTDFW [Common USB code that is used by all USB firmware ]
            │        ＋──── inc                    Common header file of USB driver
            │        ＋──── src                    USB driver
            ＋──── HwResource [Hardware access layer; to initialize the MCU ]
                       ＋──── inc                  Common header file of hardware resource
                       ＋────src                   Hardware resource
```

[Note]

a. The project for CA78K0R compiler is stored under the CS+ folder.

b. The project for KPIT GNU compiler is stored under the e^2 studio folder.

c. Refer to **13 Using the e2 studio project with CS+** section when using CC-RL compiler on CS+.

Table 3-2 shows the file structure for PDCD.

Table 3-2 File Structure

File Name	Description	Note
include/r_usb_catapi_define.h	Device driver header file	
include/r_usb_cmsc_define.h	PDCD(PMSCD+PMSDD) common header file	
include/r_usb_pmsc_api.h	PMSC API functions header file	
include/r_usb_pmsc_define.h	PMSCD header file	
include/r_usb_pmsc_extern.h	External reference header file	
MEDIA/r_usb_atapi_driver_config.h	Device driver configuration header file	
MEDIA/r_usb_atapi_driver.c	Device driver (PMSDD/media driver)	Sample Code for ATAPI
PMSC/r_usb_pmsc_ddi.c	PMSDD interface functions (DDI) Driver registration, storage command callback.	
PMSC/r_usb_pmsc_driver.c	USB class driver (PMSCF)	
PMSC/r_usb_pmsc_pci.c	PCD interface functions (PCI)	
PMSC/r_usb_pmsc_request.c	PCD interface functions (class requests)	
APL/r_usb_pmsc_descriptor.c	Mass storage class descriptor	Need to change as to user system

3.3 System Resources

There is a scheduler that invokes a “task” when it has message(s) pending in the task’s mailbox, and according to the task’s priority. Table 3-3 lists the ID and priority used to register PMSC in the scheduler. These are defined in the **r_usb_ckernelid.h** header file.

For details, refer to the Renesas USB MCU USB Basic Mini Firmware Application note.

Table 3-3 ‘Tasks’ (Mailboxes)

Object	Task Name / ID / Mailbox	Module
Task	USB_PCD_TSK / USB_TID_0	usb_pstd_pcd_task (*r_usb_pdriver.c*) Priority: USB_TID_0 (default=0)
	USB_PMSC_TSK / USB_TID_2	PMSCD, or **usb_pmsc_Task** *(r_usb_pmsc_driver.c)* Priority: USB_TID_2 (default=2)
	USB_PFLSH_TSK / USB_TID_1	PMSDD, or **usb _pmsc_SmpAtapiTask** *(r_usb_atapi_driver.c)* Priority: USB_TID_1 (default=1)
Mailbox ID	USB_PMSC_MBX / USB_PMSC_TSK	PDCD => PMSCD / PMSDD => PMSCD *(r_usb_pmsc_pci.c,* *r_usb_pmsc_driver.c,* *r_usb_pmsc_ddi.c)*
	USB_PFLSH_MBX / USB_PFLSH_TSK	PMSCD => PMSDD mailbox ID *(r_usb_atapi_driver.c)*

4. Peripheral MSC Sample Application (APL)

PMSC's main function is to enable file read/write operations for the connected USB mass storage host. The USB peripheral is to be recognized by the host (e.g. a PC) as a removable disk, so that it can perform operations such as read and write files. The Mass Storage Class specification defines the transport protocol (BOT), however, various command sets could be used to control a storage device. The following are examples of command sets which can be used over USB:

SFF-8070i, (ATAPI) * – *Command set used in this sample code.*
SFF-8020i, MMC-2 (ATAPI)
QIC-157
UFI
SCSI transparent command set

This sample mass storage device driver supports the storage command set **SFF-8070i (ATAPI)***.

* As listed in "Mass Storage Specification Overview v1.2", command block specification SFF-8070i is used (bInterfaceSubClass = 05h) together with protocol code "USB Mass Storage Class Bulk-Only" (BBB; bInterfaceProtocol = 050h).

4.1 Operating Environment

The storage media driver uses a 512K EEPROM in the default RSK cnfiguration. This EEPROM is controlled by SPI/CSI.

The EEPROM may not be mounted on your RSK board. In order to operate this PMSC sample firmware, prepare the EEPROM and any connection needed.

[Note]

CSI(Communication Serial Interface) is the interface function that RL78 series supports.

Figure 4-2 illustrates the operating environment, application operation s example.

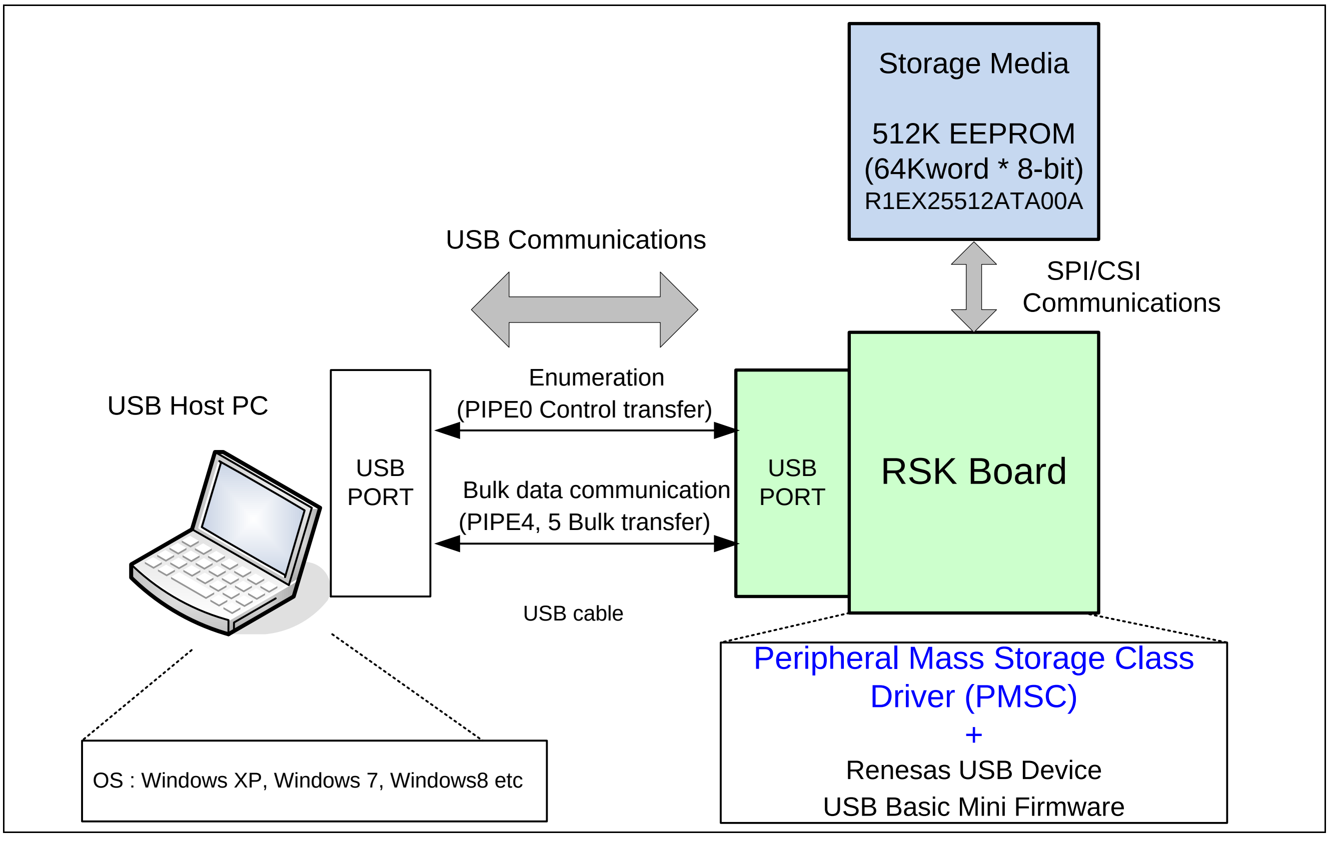

Figure 4-1 Operating Environment Example

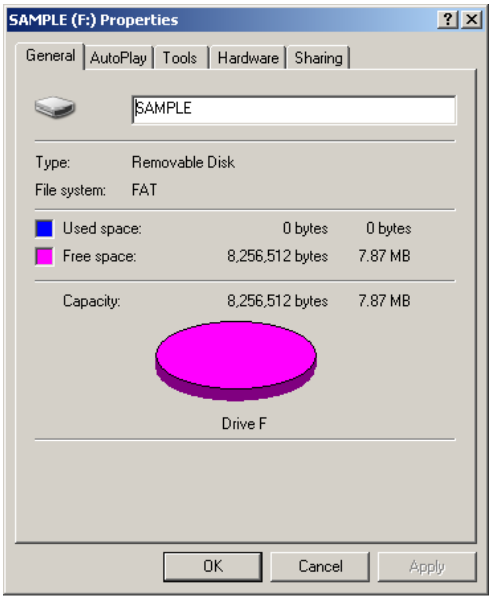

Figure 4-2 Application Operations Example. These are the properties shown for an RSK storage drive, when PMSC is connected to a PC.

Table 4-1 shows EEPROM specifics per RSK type.

Table 4-1 EEPROM connection specification

RL78/G1C			
Connection Signal	**CSI01 Signal Name**	**RSK Port/Junction Pin**	**EEPROM Pin/Number**
Clock	SCK01	P75/J1-6	C/6
Data Transfer(RL78/G1C->EEPROM)	SI01	P74/J1-7	D/5
Data Transfer(RL78/G1C<-EEPROM)	SO01	P73/J1-8	Q/2
Chip Select	--	P30/J1-12	S/1
RL78/L1C			
Connection Signal	**CSI20 Signal Name**	**RSK Port/Junction Pin**	**EEPROM Pin/Number**
Clock	SCK20	P10/J4-2	C/6
Data Transfer(RL78/L1C->EEPROM)	SI20	P11/J4-1	D/5
Data Transfer(RL78/L1C<-EEPROM)	SO20	P12/J3-25	Q/2
Chip Select	--	P30/J2-12	S/1

4.1.1 Application Program Flow

In a sense, there is no “user application” The mass storage class driver and mass storage device driver solely executes requests from the host.

Figure 4.1 shows the application processing flow overview.

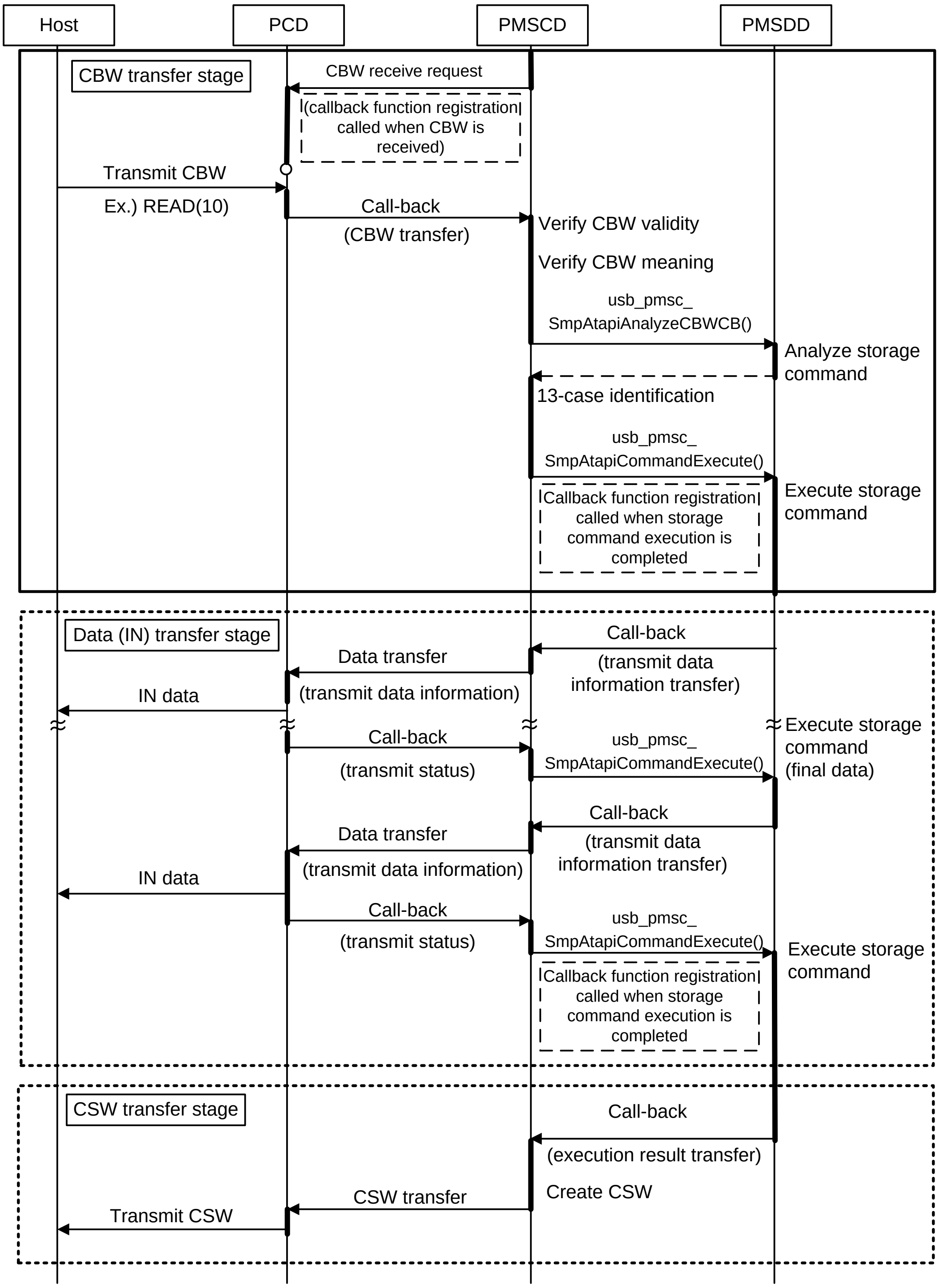

Figure 4.1 Application Processing Sequence

4.2 API tasks

Table 4-2 shows lists the APL tasks.

Table 4-2 Lists of APL tasks

Function name	Description
usb_cstd_task_start	Task(PCD, MSCD, APL) start setting
usb_pmsc_task_start	Starts a variety of tasks for the peripheral USB
usb_papl_task_start	Start Application task
usb_apl_task_switch	Switches Task

5. Peripheral Device Class Driver (PDCD)

5.1 Basic Functions

The functions of PDCD are to:

1. Respond to mass storage class requests from USB host.
2. Respond to USB host storage commands which are encapsulated in the BOT protocol (Bulk Only Transport). See below.

5.2 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 are embedded in a "Command Block Wrapper" (CBW) and the response status in a "Command Status Wrapper" (CSW).

Figure 5-1 shows an overview of how the BOT protocol progresses with command and status data flowing between USB host and peripheral.

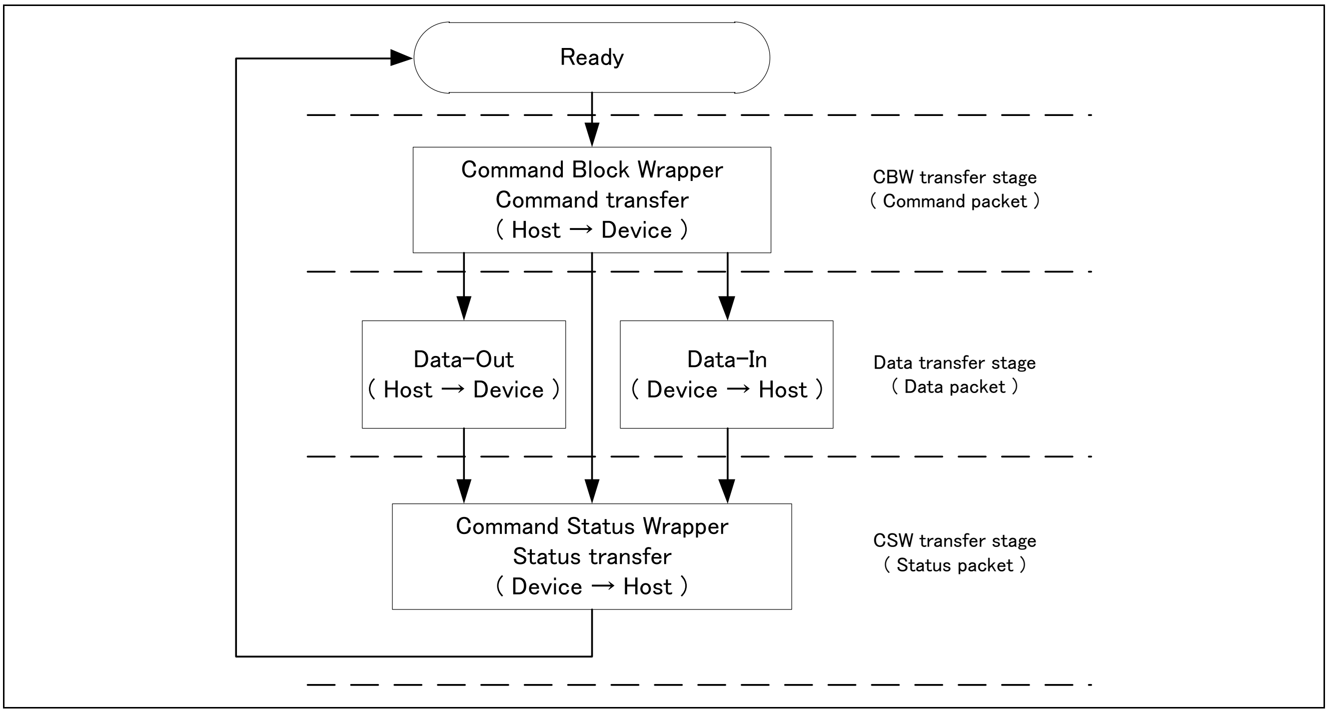

Figure 5-1 BOT protocol Overview.
Command and status flow between USB host and peripheral.

5.2.1 CBW processing

When PMSCD receives a command block wrapper (CBW) from the host, it first verifies the validity of the CBW. If the CBW is valid, PMSCD notifies PMSDD of the storage command contained in the CBW and requests analysis of PMSDD on the command. PMSCD finally performs processing based on this analysis (command validity, data transfer direction and size) and the information contained in the wrapper (data communication direction and size).

5.2.2 Sequence for storage command with *no* data transmit/receive

Figure 5-2 shows the sequence of storage commands without data transfer.

(a). CBW transfer stage

PMSCD issues a CBW receive request to PCD and registers a callback function. When PCD receives the CBW, it executes the callback which starts the CBW transfer stage. PMSCD verifies the validity of the CBW and transfers the storage command (CBWCB) to PMSDD which executes the storage command. PMSDD returns the result to PMSCD.

(b). CSW transfer stage

Based on the execution result at the time of callback, PMSCD creates a command status wrapper (CSW) and transmits it to the host via PCD.

For details on PCD operation refer to the USB Basic Mini Firmware Application note.

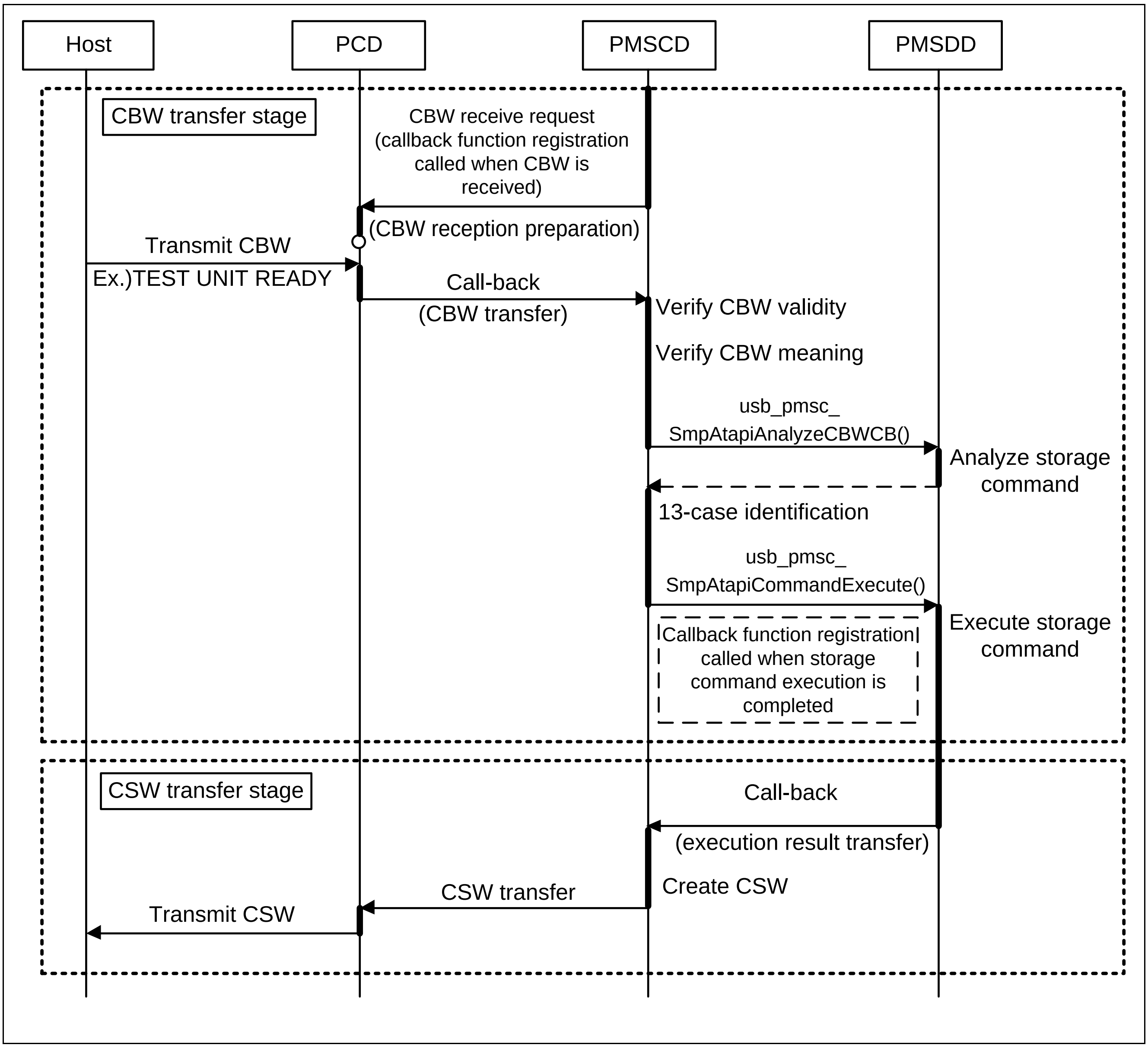

Figure 5-2 Sequence of Storage Command for No Transmit/Receive Data

5.2.3 Sequence with storage command for transmit (IN) data

Figure 5-3 shows the sequence of storage command when there is transmit (IN) data from the peripheral side.

(a). CBW transfer stage

PMSCD executes a CBW receive request to PCD, and sets up a callback. When PCD receives the CBW it executes the callback. PMSCD verifies the validity of the CBW and transfers the storage command (CBWCB) to PMSDD which analyzes the data transmit command and returns the result to PMSCD. PMSCD then reads the CBW and sends an ATAPI storage command execution request to PMSDD together with a callback registration.

(b). Data IN transfer stage

Based on the execution result at the time of callback, PMSCD notifies PCD of the data storage area and data size, and data communication with the USB host takes place. When the peripheral PCD issues a transmit end notification (status), PMSCD once again sends a continuation request to PMSDD, and data transmission is repeated.

(c). CSW transfer stage

When PMSCD receives a command processing end result from PMSDD, PMSCD creates a command status wrapper (CSW) and transmits it to the host via PCD.

For PCD operation details refer the USB Basic Mini Firmware Application note.

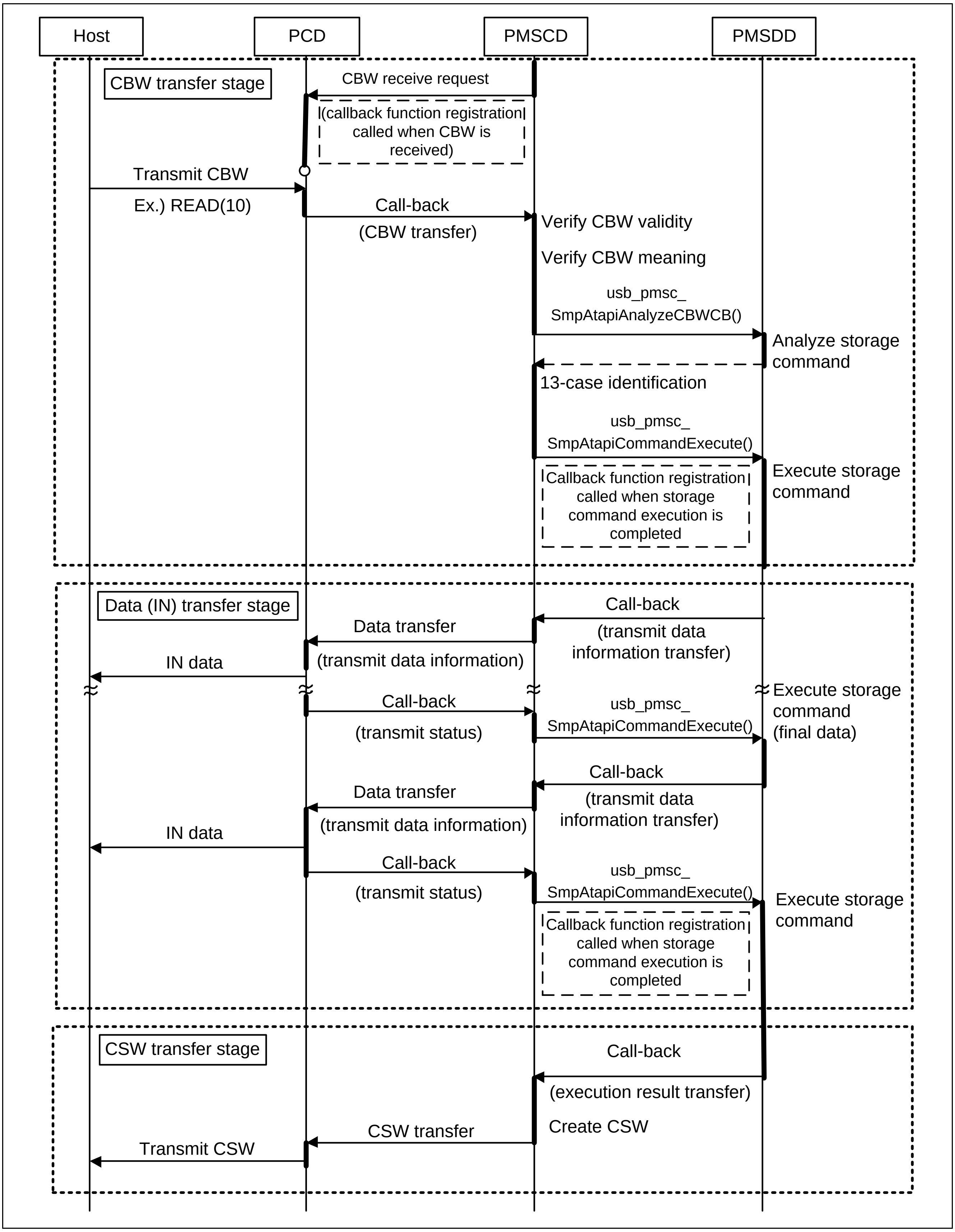

Figure 5-3 Sequence of Storage Command for Transmit (IN) Data

5.2.4 Sequence for storage command with receive (OUT) data

Figure 5-4 shows the sequence of storage command when there is transmit (OUT) data from the peripheral.

(a). CBW transfer stage

In the CBW transfer stage, PMSCD issues a CBW receive request to PCD and sets up a callback.. When PCD receives the CBW it executes the callback. PMSCD verifies the validity of the CBW and transfers the storage command (CBWCB) to PMSDD. PMSDD analyzes the data transmit command, and returns the result to PMSCD. PMSCD then compares the analysis result from PMSDD with the information contained in the CBW and sends an ATAPI storage command execution request to PMSDD together with a callback registration.

(b). Data OUT transfer stage

Based on the callback execution result, PMSCD notifies PCD of the data storage area and data size, and data communication with the host takes place. When it receives transmit end notification from PCD, PMSCD once again sends a common continuation request to PMSDD, and data transmission is repeated.

(c). CSW transfer stage

When it receives a command processing end result from PMSDD, PMSCD creates a command status wrapper (CSW) and transmits it to the host via PCD.

For PCD operation details refer to the USB Basic Mini Firmware Application note.

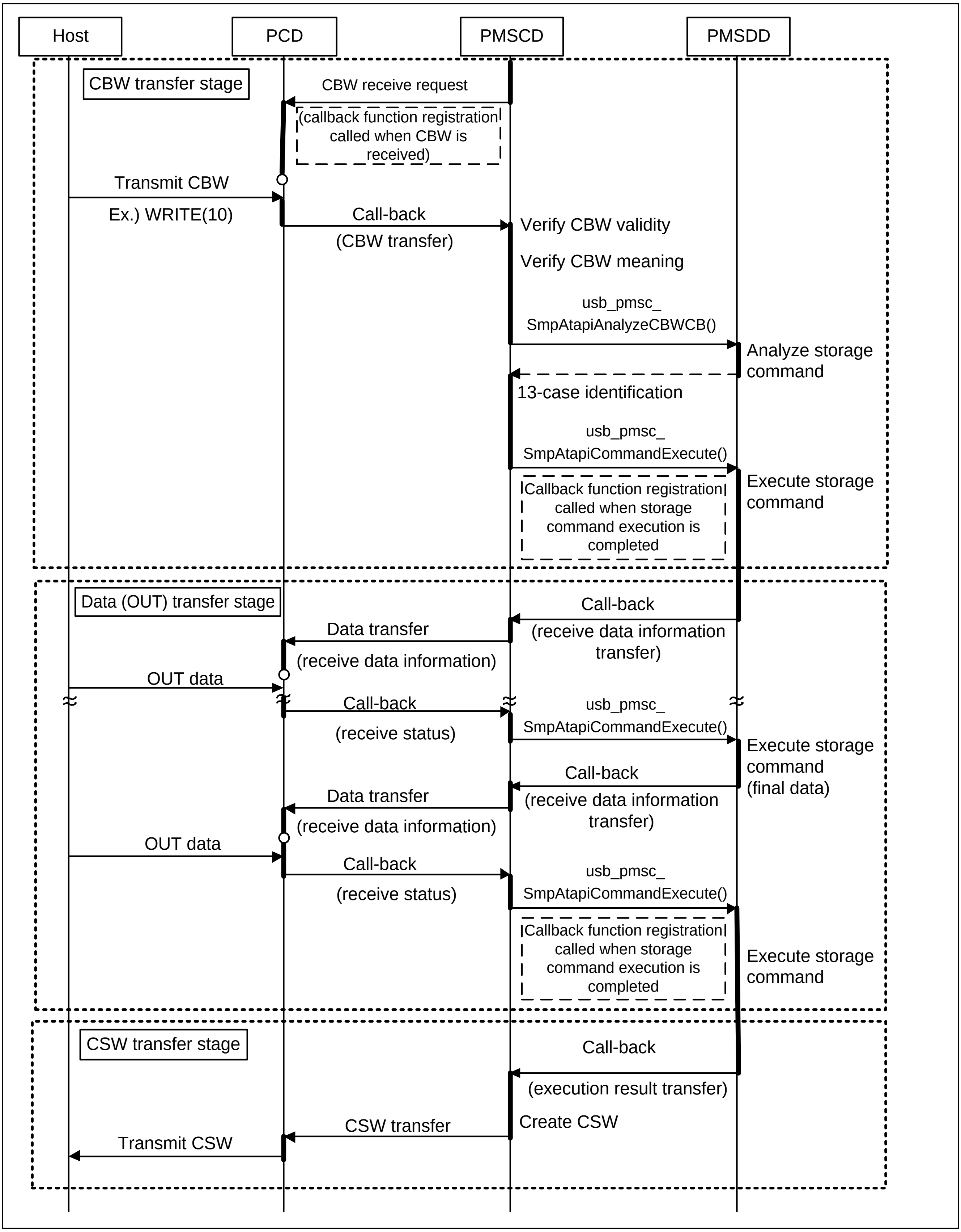

Figure 5-4 Sequence of Storage Command for Receive (OUT) Data

5.2.5 Access sequence for class request

Figure 5-5 shows the sequence when a mass storage class request is received.

(a). Setup Stage

When PCD receives a class request in the control transfer Setup Stage, it sends a request received notification to PMSCD.

(b). Data Stage

PMSCD executes the control transfer Data Stage and notifies PCD of data stage end by means of a callback function.

(c). Status Stage

PCD executes the Status Stage and ends the control transfer.

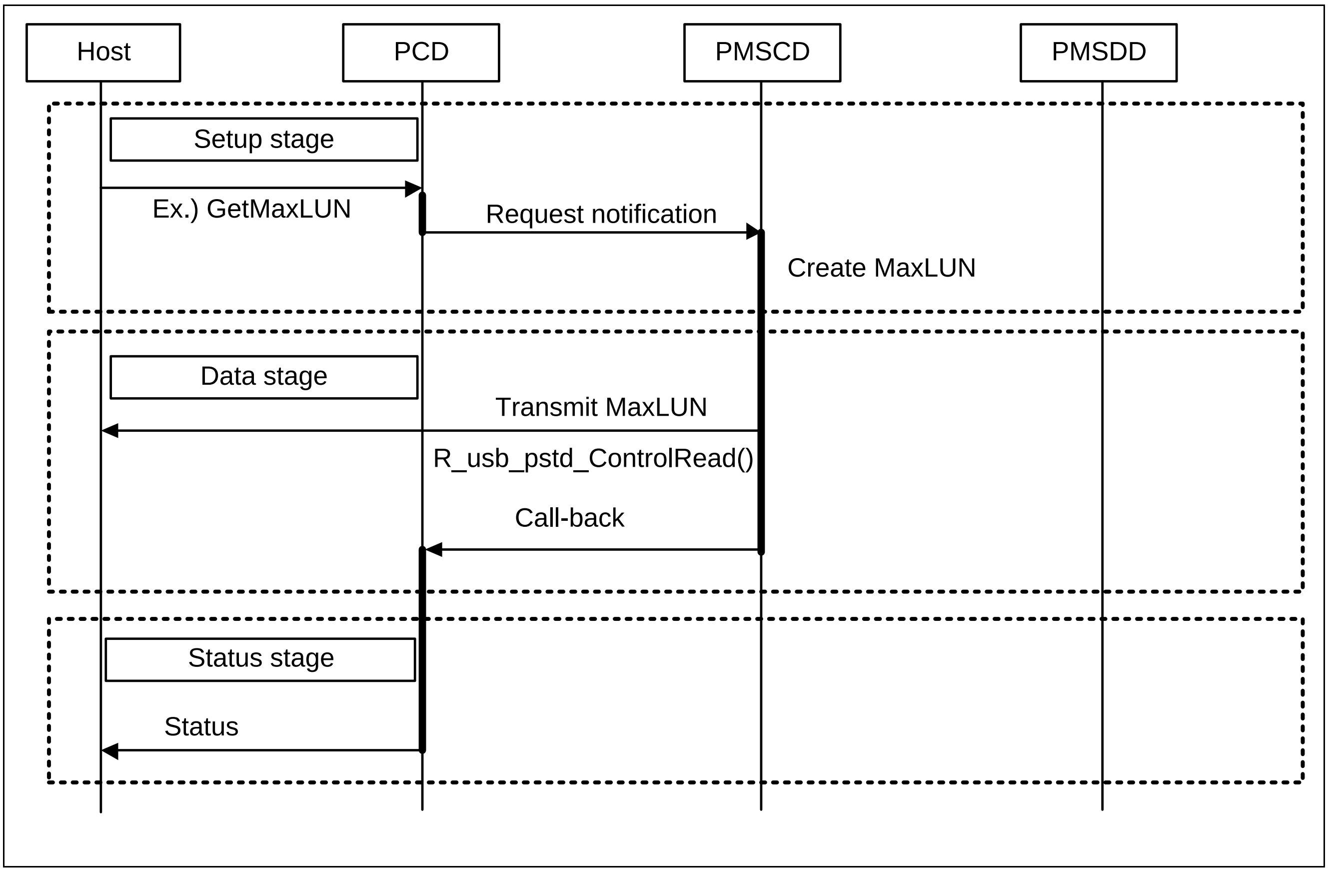

Figure 5-5 Sequence for Class Request

6. USB Peripheral Mass Storage Class Driver (PMSCD)

6.1 Basic Functions

The basic interface functions of PMSCD register, open, and close the Peripheral Mass Storage Class Driver.

The rest of the functionality inside PMSCD was described in the sequence charts in chapter 5 Peripheral Device Class Driver (PDCD).

6.2 List of API Functions

Table 6-1 List of API Functions

Function Name	Description
R_usb_pmsc_Registration	Registers PMSC driver
R_usb_pmsc_Open	Open PMSC driver
R_usb_pmsc_Close	Close PMSC driver

R_usb_pmsc_Registration

Registers PMSC driver

Format

void R_usb_pmsc_Registration(void)

Arguments

—

Return Values

—

Description

Registers the USB Peripheral Mass Storage Class driver with PCD.

Make your changes to the registration function according to the application program.

Use the *usb_pcdreg_t* type structure when registering the class driver. The information registered is as follows. For structure member details, refer to "USB Basic Mini Firmware mini Application note".

pipetbl	Pipe information table address
devicetbl	Device descriptor address
configtbl	Configuration descriptor address
stringtbl	String descriptor address table
statediagram	Callback function to start at change usb state
ctrltrans	Callback function to start at control transfer for the user

Notes

1. If a callback is not needed, register a dummy function.
2. For USB device state detail refer to "Universal Serial Bus Specification Revision 2.0 " Figure 9-1 Device State Diagram.
3. The string descriptor address table must be filled out. An example is as follows

```
uint8_t  *usb_gpmsc_StrPtr[USB_STRINGNUM] =
{
  usb_gpmsc_StringDescriptor0, /* Language ID String Descriptor Address */
  usb_gpmsc_StringDescriptor1, /* iManufacturer String Descriptor Address */
  usb_gpmsc_StringDescriptor2, /* iProduct String Descriptor Address */
  usb_gpmsc_StringDescriptor3, /* iInterface String Descriptor Address */
  usb_gpmsc_StringDescriptor4, /* iConfiguration String Descriptor Address */
  usb_gpmsc_StringDescriptor5, /* iConfiguration String Descriptor Address */
  usb_gpmsc_StringDescriptor6  /* iSerialNumber String Descriptor Address */
};
```

Example

```
void usb_pmsc_task_start( void )
{
     R_usb_pmsc_Registration();     /* Peripheral Application Registration */
     R_usb_pstd_PcdChangeDeviceState(USB_DO_SETHWFUNCTION); /* Initialize USB
HW */
   R_usb_pmsc_driver_start(); /* Peripheral Class Driver Task Start Setting */
   usb_pstd_usbdriver_start(); /* Peripheral USB Driver Start Setting */
   usb_papl_task_start();      /* Peripheral Application Task Start Setting */
}
```

R_usb_pmsc_Open

Open PMSC driver

Format

usb_er_t R_usb_pmsc_Open(uint16_t data1, uint16_t data2)

Argument

uint16_t	data1	: Not used
uint16_t	data2	: Not used

Return Value

uint16_t Processing result. 0 = USB_E_OK

Description

This function is to be called when the USB device has been connected, has enumerated and been configured by the USB host .

The function sets the CBW reception setting.

Note

Call this function in the callback function which is registered in the structure(*usb_pcdreg_t*) member (*statediagram*)

Example

```
void usb_pmsc_change_device_state( uint16_t data1, uint16_t device_state )
{
    switch ( device_state )
    {
    case USB_STS_CONFIGURED:
        R_usb_pmsc_Open(data1, device_state);
    break;
    }
}
```

R_usb_pmsc_Close

Close PMSC driver

Format

Usb_er_t R_usb_pmsc_Close(Usb_utr_t *ptr, uint16_t data1, uint16_t data2)

Argument

Usb_utr_t	*ptr	: Pointer to a USB Transfer Structure	
uint16_t		data1	: Not used
uint16_t		data2	: Not used

Return Value

uint16_t — USB_E_OK

Description

This function is to be called when the USB device has been disconnected..

This function is called at transition to the detached state. There are no operations. Add if necessary.

Note

This function should be called when the callback function registered with API *R_usb_pmsc_Registration()* is triggered for state change USB_STS_DETACH. See below.

Example

```
void usb_pmsc_change_device_state( uint16_t data1, uint16_t device_state )
{
    switch ( device_state )
    {
    case USB_STS_DETACH:
        R_usb_pmsc_Close(data1, device_state);
    break;
    }
}
```

6.3 Class Driver Registration

The device class driver PMSCD must be registered with PCD to function. Use the *R_usb_pmsc_Registration()* function to register PMSCD, using the sample code as reference. See 8.1, For details, refer to the Renesas USB MCU USB Basic Firmware mini Application note.

6.4 User Definition Tables

It is necessary to create a descriptor table and Pipe Information Table for use by PCD. Refer to the sample file *r_usb_pmsc_descriptor.c* when creating these tables. For details on the Pipe Information Table refer to the Renesas USB MCU USB Basic Firmware mini Application note.

7. Peripheral Mass Storage Device Driver (PMSDD)

The main function of PMSDD is to analyze and call for execution of storage commands received from the host via PMSCD. Host enumeration is determined by the *InterfaceSubClass* as set in the descriptor source code and is set to SFF-8070i (ATAPI). This command set is used by the host to control the storage media. These are the storage commands:

READ10
INQUIRY
REQUEST_SENSE
MODE_SENSE6
MODE_SENSE10
READ_FORMAT_CAPACITY
READ_CAPACITY
WRITE10
WRITE_AND_VERIFY
MODE_SELECT6
MODE_SELECT10
FORMAT_UNIT
TEST_UNIT_READY
START_STOP_UNIT
SEEK
VERIFY10
PREVENT_ALLOW

PMSDD notifies PMSCD of communication data and execution results related to storage command execution.

PMSDD divides the data transfer intp0 pieces when the transfer data length exceeds the user-specified block count.

A master boot record (FAT12) sample table is provided.

7.1 PMSDD Storage Command Structure

The "storage command structure" is *USB_PMSC_CDB_t*. The format of a storage command (SFF-8070i) differs depending on the command category, so a union is used for the command. Four patterns encompass ten command categories as shown in Table 7-1.

Table 7-1 USB_PMSC_CDB_t Structure

Union Member	Type	Structure Member	Bit Count	Command Category
s_usb_ptn0	uint8_t	uc_OpCode		Command determination (common)
	uint8_t s_LUN	b_LUN	3	
		b_reserved	5	
	uint8_t	uc_data		
s_usb_ptn12	uint8_t	uc_OpCode		INQUIRY / REQUEST_SENSE
	uint8_t s_LUN	b_LUN	3	
		b_reserved4	4	
		b_immed	1	
	uint8_t	uc_rsv2[2]		
	uint8_t	uc_Allocation		
	uint8_t	uc_rsv1[1]		
	uint8_t	uc_rsv6[6]		
s_usb_ptn378	uint8_t	uc_OpCode		Not used (FORMAT UNIT)
	uint8_t s_LUN	b_LUN	3	
		b_FmtData	1	
		b_CmpList	1	
		b_Defect	3	
	uint8_t	ul_LBA0		
	uint8_t	ul_LBA1		
	uint8_t	ul_LBA2		
	uint8_t	ul_LBA3		
	uint8_t	uc_rsv6[6]		
s_usb_ptn4569	uint8_t	uc_OpCode		READ10 / WRITE10 / WRITE _AND_VERIFY / MODE_SENSE / FORMAT CAPACITY / MODE SELECT
	uint8_t s_LUN	b_LUN	3	
		b_1	1	
		b_reserved2	2	
		b_ByteChk	1	
		b_SP	1	
	uint8_t	ul_LogicalBlock0		
	uint8_t	ul_LogicalBlock1		
	uint8_t	ul_LogicalBlock2		
	uint8_t	ul_LogicalBlock3		
	uint8_t	uc_rsv1[1]		
	uint8_t	us_Length_Hi		
	uint8_t	us_Length_Lo		
	uint8_t	uc_rsv3[3]		

Table 7-2 shows storage commands analysis result.

Table 7-2 The USB_PMSC_CBM_t Structure
Contains "analysis" result of usb_pmsc_SmpAtapiAnalyzeCbwCb.

Type	Member	PMSDD storage command analysis RESULT	Remarks
uint32_t	ar_rst	Data direction.	Direction of data transported in last ATAPI command.
uint32_t	ul_size	Data size	Size of data in last ATAPI command.

7.2 List of PMSDD Functions

Table 7-3 lists the functions of PMSDD.

Table 7-3 List of PMSDD Functions

Function Name	Description
usb_pmsc_SmpAtapiAnalyzeCbwCb	Analyzes storage command.
usb_pmsc_SmpAtapiTask	Main task of PMSDD
usb_pmsc_SmpAtapiInitMedia	Initialization at PMSDD start
usb_pmsc_SmpAtapiCloseMedia	Processing at PMSDD end
usb_pmsc_SmpAtapiCommandExecute	Transmits message from PMSCD to PMSDD main task.

7.3 PMSDD Task Description

PMSDD receives storage commands from PMSCD and executes them. PMSDD also receives host data transfer results from PMSCD. Table 7-4 lists PMSDD command processing. When the transfer data size exceeds USB_ATAPI_BLOCK_UNIT, the data is divided into smaller units and transferred.

For commands that do not involve memory access, the transmitted data is created from the response data tables
g_pmsc_atapi_data_size[],
g_pmsc_atapi_rd_dat_idx[],
g_pmsc_atapi_req_idx[], and
g_pmsc_atapi_rd_dat_tbl[]. (*1)

(*1) These response data tables follow storage command set SFF-8070i. The index into the table is determined by the command. Refer to *uc_OpCode* in Table 7-1 USB_PMSC_CDB_t Structure.

Table 7-4 Corresponding Function per Storage Command

Storage command	Corresponding Function	Description
READ10	pmsc_atapi_get_read_memory()	Gets start address and size.
INQUIRY	pmsc_atapi_get_read_data()	Selects response data from array g_pmsc_atapi_rd_dat_tbl.
REQUEST_SENSE	pmsc_atapi_get_read_data ()	Selects response data from array g_pmsc_atapi_rd_dat_tbl.
MODE_SENSE10	pmsc_atapi_get_read_data ()	Selects response data from array g_pmsc_atapi_rd_dat_tbl.
READ_FORMAT_CAPACITY	pmsc_atapi_get_read_data ()	Selects response data from array g_pmsc_atapi_rd_dat_tbl.
READ_CAPACITY	pmsc_atapi_get_read_data ()	Selects response data from array g_pmsc_atapi_rd_dat_tbl.
WRITE10	pmsc_atapi_get_write_memory()	Gets start address and size.
WRITE_AND_VERIFY	pmsc_atapi_get_write_memory()	Gets start address and size.
MODE_SELECT10	pmsc_atapi_get_write_memory()	Gets start address and size.
FORMAT_UNIT	pmsc_atapi_get_write_memory()	Gets start address and size.
TEST_UNIT_READY	usb_pmsc_SmpAtapiTask()	Status = USB_PMSC_CMD_COMPLETE
START_STOP_UNIT	usb_pmsc_SmpAtapiTask()	Status = USB_PMSC_CMD_COMPLETE
SEEK	usb_pmsc_SmpAtapiTask()	Status = USB_PMSC_CMD_COMPLETE
VERIFY10	usb_pmsc_SmpAtapiTask()	Status = USB_PMSC_CMD_COMPLETE
PREVENT_ALLOW	usb_pmsc_SmpAtapiTask()	Status = USB_PMSC_CMD_FAILED
others	usb_pmsc_SmpAtapiTask()	Status = USB_PMSC_CMD_ERROR

8. Media Driver Interface

This chapter is an introduction to the block USB Peripheral Mass Storage Class Driver (PMSC), and how it is used for PMSC. For complete details on this API and how to create new media drivers that interface through it, see application note no. r01an1443eu_rx.

PMSC is able to operate with a variety of devices as data storage media. It uses a block storage type driver API described in the application note (Document No. r01an1443eu_rx). The storage media interface is an abstract set of functions (*R_MEDIA_Read*, *R_MEDIA_Write*, etc) which are the same regardless of the underlying driver that will be called behind the interface. PMSC can interface any media driver that supports this API.

8.1 Overview of Media Driver API Functions

The Block Access Media Driver API serves to interface the PMSC application to a specific media device driver. The selection of media is made through configuration files that the user must customize. There is one configuration file for the Block Access Media Driver API, *r_media_driver_api_config.h*, which has a list of media devices, and another configuration file for PMSC, *r_usb_atapi_driver_config.h*, which assigns the selected media driver to be used for PMSC.

The transport layer subtype in this application is SFF-8070i (ATAPI). This layer processes the storage commands that are contained in the Command Blocks that are tunneled through the BOT transport layer. Most of the work done to process the command set is accomplished by routines in the file *r_usb_atapi_driver.c*. This is where the ATAPI data storage commands that write or read the storage media, that is, the block API calls, originate. Storage commands pass through the Block Access Media Driver API layer where they are directed to drivers for the assigned storage device.

Table 8-1 The Block Access Media Driver API functions

Function Name	Description
R_MEDIA_Initialize	Registers the media driver
R_MEDIA_Open	Open media driver
R_MEDIA_Close	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

8.2 Selecting Media Driver

A media driver has a structure that contains the pointers to its implementation of the API's abstract functions shown in Table 8-1. The name of this driver implementation structure must be assigned to a macro used by the ATAPI task: ATAPI_MEDIA_DEVICE_DRIVER in *r_media_driver_api_config.h*.

The section, or block, size must match both what the host can handle (e.g. Windows FAT and what the bottom layer driver can handle, e.g. 512 or 4096 bytes.

8.2.1 Initializing the Media Driver Function Set

Once the block media driver functions listed in the g_MediaDriverList actually exist, all that is needed is to call the abstract function

```
R_MEDIA_Initialize(&ATAPI_MEDIA_DEVICE_DRIVER);
```

which will write the actual driver member functions to g_MediaDriverList at runtime. Once the member functions have popluated g_MediaDriverList, calls to the other abstract block media functions; R_MEDIA_Open, R_MEDIA_Read, R_MEDIA_Write,… will redirect to call the user's particular driver functions. In other words, it all happens behind the scene without the user having to replace the abstract call with the actual driver calls.

This initialization call is already done in PMSC in file *r_media_driver_api.c*.

This runtime registration of the drivers can be omitted, and the member functions be called directly, but the member functions must then <u>*not*</u> *be declared (and defined) as static in the driver source code, as in the RAM and API sample drivers.*

8.3 Changing (adding) Storage Media

Suppose you would like to change what media the data is stored on. Default is EEPROM. To use a different storage media, the read, write, etc functions must first be made to conform to the block media driver API described above.

8.3.1 Steps to conform a driver to the block media API

1. Add the media driver interface function source code using return types and arguments as specified in *r_media_driver_api.h*. Then add a *media_driver_s* structure containing pointers to these members at the top of this file.
2. In *r_usb_atapi_driver_config.h,* add the definitions as described in 8.2. and call the abstract initialize function in 8.2.1, before any calls to the block driver API functions are made.

9. The EEPROM Media Driver

The EEPROM media driver that is supplied preconfigured to use EEPROM as a storage media device. While this is provided primarily as a simplified example of a media driver to demonstrate the functionality of USB-PMSC, it can still be useful as a means to transfer data to or from the MCU to the USB host.

(1). Operates as the attached media device

The sample code operates as media with 64KByte EEPROM (Renesas: R1EX25512ATA00A).

(2). Media is preformatted to appear as removable FAT file storage device.

(3). Media driver supports connection to Windows OS (XP, 7, etc) host.

(4). Default format may be overwritten by format command from the host.

(5). Host can read files from EEPROM or write files to it, and can update the FAT format information as needed.

9.1 EEPROM Media Driver Default FAT Format

The EEPROM media device is preformatted to appear as removable FAT file storage device.

The file *r_eeprom_disk_format_data.c* contains the section declarations and the pre-initialized data that will get stored to EEPROM at system startup.

The beginning of the EEPROM area (lowest memory address) is considered to be the boot sector (sector 0), as block zero is the default boot sector area in a FAT formatted storage device. Therefore, all storage blocks are addressed relative to this location. When a host device accesses the media device it will always communicate in terms of starting logical block number “LBN”, and “block count”, the number of blocks to transfer (512 for WindowsXP and 4096 for Windows7). The host knows how to navigate a FAT formatted storage device, and will read the first sector to gain information about the specific format on the EEPROM, and then discover where to look for additional file information. From that the host will know which block numbers to access for EEPROM data storage.

Alternatively, the host can re-format the EEPROM, replacing the default boot sector, FAT tables, etc, with its own format. In this case the host still knows where in the EEPROM a specific block of data resides.

Note: It is not strictly necessary for a USB-PMSC device to have a FAT file system format, however most host systems will expect to use the PMSC device for file storage with FAT as the file system type.

The USB device cannot access the FAT storage on its own. See chapter 11.

9.2 EEPROM Global Area Variables

The entire EEPROM RAM section is of global scope with a number of named variables. Table 9-1 lists the global area variables of the media driver.

Table 9-1 Media Driver Global Areas

Type	Variable Name	Description
uint8_t	eeprom_boot_sector	Primary Boot Record area (sector 0)
uint8_t	usb_gpmsc_Table1	Dummy area (sector 1)
uint8_t	usb_gpmsc_TableFat	FAT table (sector 2 and 3)
uint8_t	usb_gpmsc_RootDir	Directory entry area (sector 4)

9.3 Constant Definitions

Table 9-2 shows the constant definitions used for the EEPROM media driver.

Table 9-2 PMSDD Constant Definitions

Description	Definition name	Value	Remark
Media type	EEPROM_MEDIATYPE	0xF8u	Modifiable
Signature	EEPROM_SIGNATURE	0xAA55u	Not modifiable
Sector size	EEPROM_SECTSIZE	512ul	Modifiable
Cluster size	EEPROM_CLSTSIZE	0x01u	Modifiable
FAT number	EEPROM_FATNUM	0x02u	Modifiable
Media size *1	EEPROM_MEDIASIZE	64*1024（=64Kbyte）	Modifiable
Total number of sectors*2	EEPROM_TOTALSECT	EEPROM_MEDIA_SIZE / EEPROM_SECTSIZE	Not modifiable
FAT Table Length*2	EEPROM_FATLENGTH	341ul（FAT12）	Not modifiable
FAT table length*2	EEPROM_FATSIZE	(((EEPROM_TOTALSECT-8) / EEPROM_FATLENGTH)+1)	Not modifiable
Root directory	EEPROM_ROOTTOP	(((EEPROM_FATSIZE * EEPROM_FATNUM+1)/8+1)*8)	Not modifiable (Not used)
FAT start	EEPROM_FATTOP	(EEPROM_ROOTTOP - (EEPROM_FATSIZE * EEPROM_FATNUM))	Not modifiable (Not used)
Root Directory size	EEPROM_ROOTSIZE	1ul	Not modifiable (Not used)

*1 A minimum 20K byte capacity is required when connecting the device to a PC running WindowsXP.

FAT12 is selected when the media size is set to under 2M bytes.

FAT16 is selected when the media size is set to under 32M bytes.

*2 Total number of sectors, FAT data length, and FAT table length are automatically calculated based on the media size.

9.4 Operation Overview

Table 9-3 lists the EEPROM media variables and Figure 9.1 shows the EEPROM media block diagram.

Table 9-3 Media Variables

Category	Sector No.	Physical Address	Accesible Size
PBR	Sector 0	0x0000	512Byte
Dummy area	Sector 1	0x0200	512Byte
FAT1	Sector 2	0x0400	512Byte× EEPROM_FATSIZE
FAT2	Sector 3	0x0600	512Byte× EEPROM_FATSIZE
ROOT DIR	Sector 4	0x0800	512Byte×16

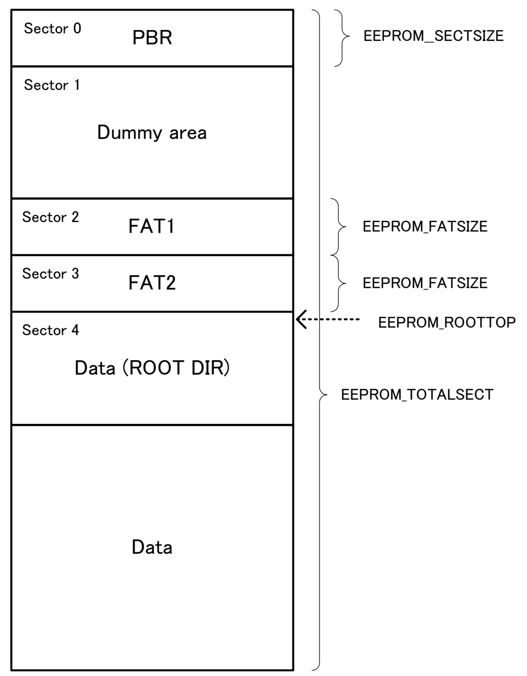

Figure 9.1 Media Block

10. Resource Registration in Scheduler

When using the scheduler, it is necessary to register resources such as task IDs, and mailbox IDs in the file"r_usb_ckernelid.h".

In the sample file, the registrations are as follows.

```
/* Peripheral MSC Sample Task */
#define  USB_PFLSH_TSK     USB_TID_1      /* Task ID */
#define  USB_PFLSH_MBX     USB_PFLSH_TSK  /* Mailbox ID */

/* Peripheral MSC Driver Task */
#define  USB_PMSC_TSK   USB_TID_2         /* Task ID */
#define  USB_PMSC_MBX   USB_PMSC_TSK      /* Mailbox ID */
```

11. Limitations

The PMSC firmware does not include a FAT library, so it cannot access the storage via file system calls on its own. Adding local media access is possible, but a FAT driver must be added to the firmware.

12. Setup for the e^2 studio project

(1). Start up e^2 studio.

* If starting up e^2 studio for the first time, the Workspace Launcher dialog box will appear first. Specify the folder which will store the project.

(2). Select [File] → [Import]; the import dialog box will appear.

(3). In the Import dialog box, select [Existing Projects into Workspace].

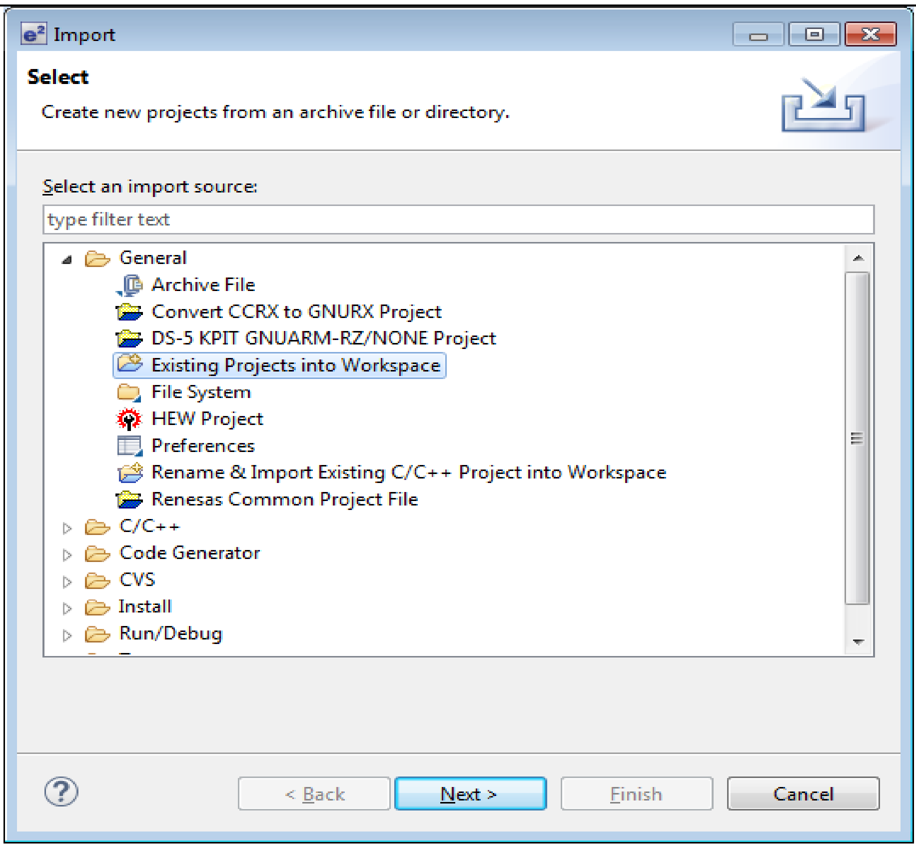

Figure 12-1 Select Import Source

(4). Press [Browse] for [Select root directory]. Select the folder in which [.cproject] (project file) is stored.

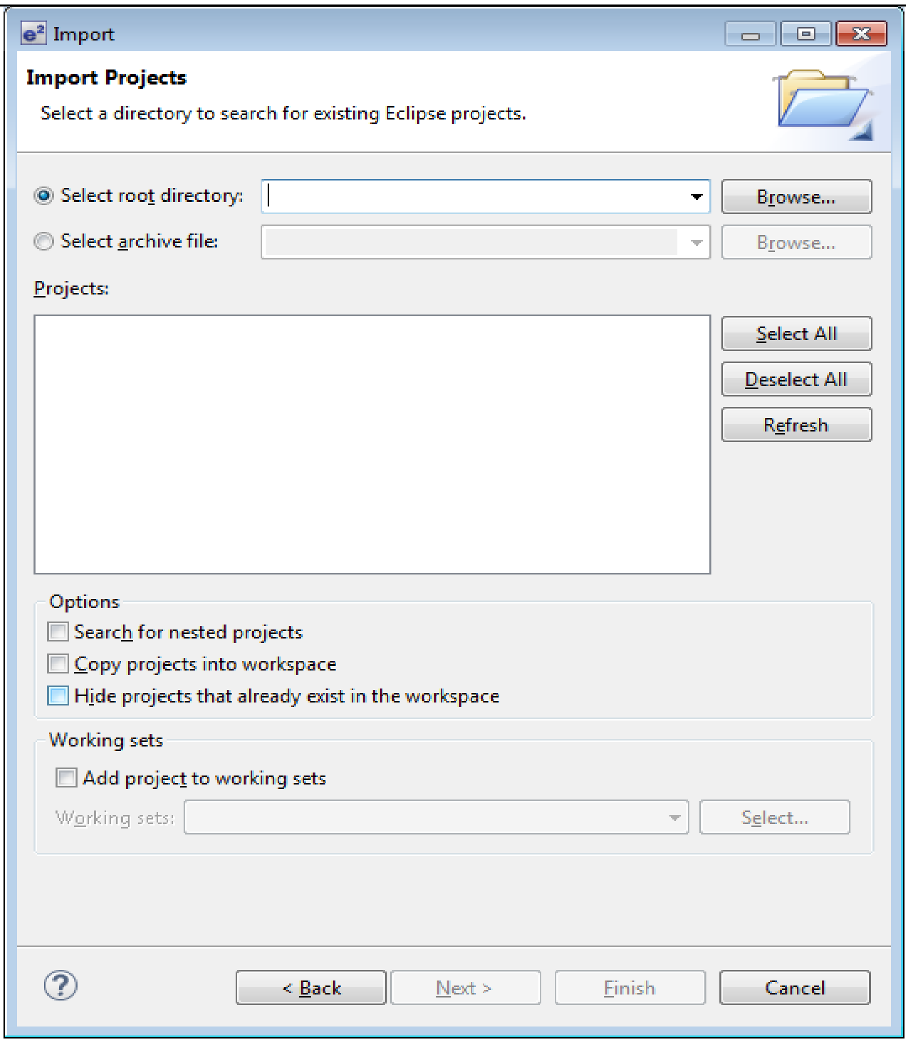

Figure 12-2 Project Import Dialog Box

(5). Click [Finish].

This completes the step for importing a project to the project workspace.

13. Using the e² studio project with CS+

This package contains a project only for e² studio. When you use this project with CS+, import the project to CS+ by following procedures.

[Note]

The *rcpc* file is stored in "workspace\RL78\CCRL*devicename*" folder.

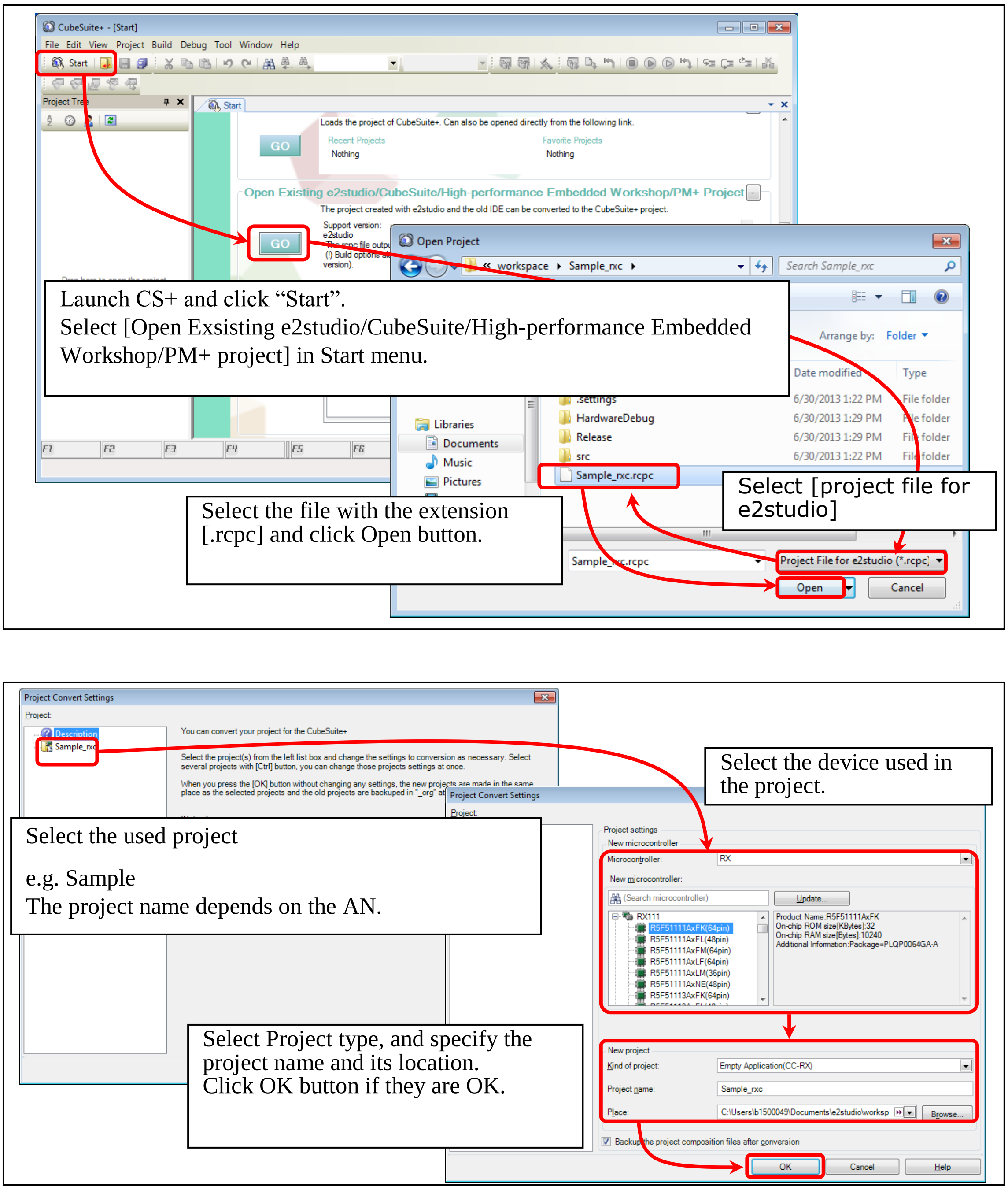

Figure 13-1 Using the e² studio project with CS+

Website and Support

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

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

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Dec.10.2010	—	First edition issued
2.00	Mar.29.2013	—	Revision of the document by firmware upgrade. Complete rewrite of Media Driver chapter.
2.10	Aug.01.2013	—	RL78/L1C, RX111 is supported. Error is fixed.
2.11	Oct. 31. 2013	—	2.2.1 Folder Structure was corrected.
2.12	Mar.31.2014	—	R8C is supported. Error is fixed.
2.13	Mar.16.2015	—	RX111 is deleted from Target Device
2.14	Jan.18.2016	—	Supported Technical Update (Document No. TN-RL*-A055A/E and TN-RL*-A033B/E)
2.15	Mar. 28. 2016	—	CC-RL compiler is supported.

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 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 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.

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.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, 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 1207, 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 India Pvt. Ltd.
No.777C, 100 Feet Road, HAL II Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 5.0