Renesas USB MCU

R01AN2025EJ0121
Rev.1.21
Mar 31, 2017

USB Basic Host and Peripheral Driver using Firmware Integration Technology

Introduction

This application note describes the USB basic firmware, which utilizes Firmware Integration Technology (FIT). This module performs hardware control of USB communication. It is referred to below as the USB-BASIC-F/W FIT module.

Target Device

RX63N/RX631 Group
RX65N/RX651 Group
RX64M Group
RX71M Group

When using this application note with other Renesas MCUs, careful evaluation is recommended after making modifications to comply with the alternate MCU.

Related Documents

1. Universal Serial Bus Revision 2.0 specification
   【http://www.usb.org/developers/docs/】
2. RX63N/RX631 Group User’s Manual: Hardware (Document number .R01UH0041EJ)
3. RX64M Group User’s Manual: Hardware (Document number .R01UH0377EJ)
4. RX65N/RX651 User's Manual: Hardware (Document number .R01UH0590EJ)
5. RX71M Group User’s Manual: Hardware (Document number .R01UH0493EJ)

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

USB Devices Page
【http://www.renesas.com/prod/usb/】

Contents

1. Overview

The USB-BASIC-F/W FIT module performs USB hardware control. The USB-BASIC-F/W FIT module operates in combination with one type of sample device class drivers provided by Renesas.

This module supports the following functions.

<Overall>

- Supporting USB Host or USB Periphral.
- Device connect/disconnect, suspend/resume, and USB bus reset processing.
- Control transfer on pipe 0.
- Data transfer on pipes 1 to 9. (Bulk or Interrupt transfe)

<Host mode>

- In host mode, enumeration as Low-speed/Full-speed/Hi-speed device (However, operating speed is different by devices ability.)
- Transfer error determination and transfer retry.

<Peripheral mode>

- In peripheral mode, enumeration as USB Host of USB1.1/2.0/3.0.

1.1 Note

1. This application note is not guaranteed to provide USB communication operations. The customer should verify operations when utilizing the USB device module in a system and confirm the ability to connect to a variety of different types of devices.
2. The terms “USB0 module” and “USB1 module” used in this document refer to different modules for each MCU. The following is a reference.

	MCU	USB Module Name
USB0 module (Start address: 0xA0000)	RX63N/RX631	USBa module
	RX65N/RX651	USBb module
	RX64M	USBb module
	RX71M	USBb module
USB1 module (Start address: 0xA0200/0xD0400)	RX63N/RX631	USBa module
	RX64M	USBA module
	RX71M	USBAa module

Note:

The RX65N/RX651 MCU does not support the USB1 module.

1.2 Limitations

This driver is subject to the following limitations.

1. In USB host mode, the module does not support suspend/resume of the connected hub or devices connected to the hub’s down ports.
2. In USB host mode, the module does not support suspend during data transfer. Execute suspend only after confirming that data transfer is complete.
3. Multiconfigurations are not supported.
4. The USB host and USB peripheral modes cannot operate at the same time.
5. When using the USB hub for DTC/DMA transfer, only the first USB device connected to the USB hub will be able to send data using DTC/DMA transfer. All subsequent data transfers will be implemented with the CPU transfer function.
6. This USB driver does not support the error processing when the out of specification values are specified to the arguments of each function in the driver.
7. In the case of Vendor class, the user can not use the USB Hub.
8. This driver does not support the CPU transfer using D0FIFO/D1FIFO register.

1.3 Terms and Abbreviations

APL	:	Application program
CDP	:	Charging Downstream Port
DCP	:	Dedicated Charging Port
HBC	:	Host Battery Charging control
HCD	:	Host control driver of USB-BASIC-F/W
HDCD	:	Host device class driver (device driver and USB class driver)
HUBCD	:	Hub class sample driver
H/W	:	Renesas USB deviceRenesas USB MCU
MGR	:	Peripheral device state maneger of HCD
PBC	:	Peripheral Battery Charging control
PCD	:	Peripheral control driver of USB-BASIC-F/W
PDCD	:	Peripheral device class driver (device driver and USB class driver)
RSK	:	Renesas Starter Kits
STD	:	USB-BASIC-F/W
USB	:	Universal Serial Bus
USB-BASIC-F/W	:	USB Basic Host and Peripheral firmware for Renesas USB MCU
Scheduler	:	Used to schedule functions, like a simplified OS.
Task	:	Processing unit

1.4 USB-BASIC-F/W FIT module

User needs to integrate this module to the project using r_bsp. User can control USB H/W by using this module API after integrating to the project.

1.5 Software Configuration

In peripheral mode, USB-BASIC-F/W comprises the peripheral driver (PCD), and the application (APL). PDCD is the class driver and not part of the USB-BASIC-F/W. See Table 1-1. In host mode, USB-BASIC-F/W comprises the host driver (HCD), the manager (MGR), the hub class driver (HUBCD) and the application (APL). HDD and HDCD are not part of the USB-BASIC-F/W, see Table 1-1
The peripheral driver (PCD) and host driver (HCD) initiate hardware control through the hardware access layer according to messages from the various tasks or interrupt handler. They also notify the appropriate task when hardware control ends, of processing results, and of hardware requests.
Manager manages the connection state of USB peripherals and performs enumeration. In addition, manager issues a message to host driver or hub class driver when the application changes the device state. Hub class driver is sample program code for managing the states of devices connected to the down ports of the USB hub and performing enumeration.
The customer will need to make a variety of customizations, for example designating classes, issuing vendor-specific requests, making settings with regard to the communication speed or program capacity, or making individual settings that affect the user interface.

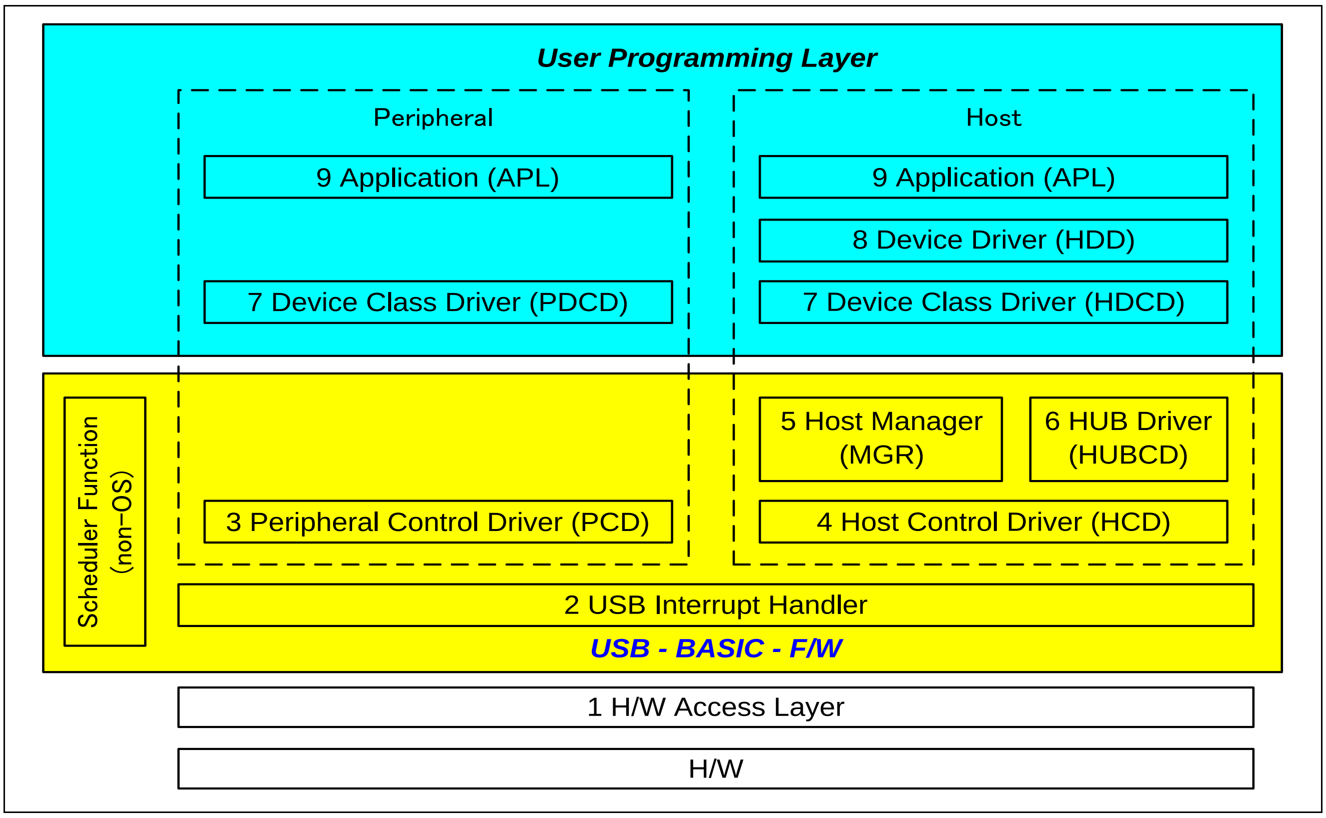

Figure 1-1 Task Configuration of USB-BASIC-F/W

Table 1-1 Software function overview

No	Module Name	Function
1	H/W Access Layer	Hardware control
2	USB Interrupt Handler	USB interrupt handler (USB packet transmit/receive end and special signal detection)
3	Peripheral Control Driver (PCD)	Hardware control in peripheral mode Peripheral transaction management
4	Host control driver (HCD)	Hardware control in host mode Host transaction management
5	Host Manager (MGR)	Device state management Enumeration HCD/HUBCD control message determination
6	HUB Driver (HUBCD)	HUB down port device state management HUB down port enumeration
7	Device Class Driver	Provided by the customer as appropriate for the system.
8	Device Driver	Provided by the customer as appropriate for the system.
9	Application	Provided by the customer as appropriate for the system.

1.6 Scheduler Function and Tasks

When using the non-OS version of the source code, a scheduler function manages requests generated by tasks and hardware according to their relative priority. When multiple task requests are generated with the same priority, they are executed using a FIFO configuration. To assure commonality with non-OS firmware, requests between tasks are implemented by transmitting and receiving messages.

1.7 Pin Setting

Do the pin setting used in thie module before calling *R_USB_Open* function.

2. Peripheral

2.1 Peripheral Control Driver (PCD)

2.1.1 Basic functions

PCD is a program for controlling the hardware. PCD analyzes requests from PDCD (not part of the USB-BASIC-F/W FIT module) and controls the hardware accordingly. It also sends notification of control results using a user provided call-back function. PCD also analyzes requests from hardware and notifies PDCD accordingly.

PCD accomplishes the following:

1. Control transfers. (Control Read, Control Write, and control commands without data stage.)
2. Data transfers. (Bulk, interrupt) and result notification.
3. Data transfer suspensions. (All pipes.)
4. USB bus reset signal detection and reset handshake result notifications.
5. Suspend/resume detections.
6. Attach/detach detection using the VBUS interrupt.
7. Hardware control when entering and returning from the clock stopped (low-power sleep mode) state.

2.1.2 Issuing requests to PCD

API functions are used when hardware control requests are issued to the PCD and when performing data transfers.

Refer to chapter **4, API Functions** for the API function.

2.1.3 USB requests

This driver supports the following standard requests.

1. GET_STATUS
2. GET_DESCRIPTOR
3. GET_CONFIGURATION
4. GET_INTERFACE
5. CLEAR_FEATURE
6. SET_FEATURE
7. SET_ADDRESS
8. SET_CONFIGURATION
9. SET_INTERFACE

This driver answers requests other than the above with a STALL response.

Note that, refer to chapter **9, USB Class Requests** for the processing method when this driver receives the class request or vendor request.

2.2 API Information

This Driver API follows the Renesas API naming standards.

2.2.1 Hardware Requirements

This driver requires your MCU support the following features:

- USB

2.2.2 Software Requirements

This driver is dependent upon the following packages:

- r_bsp
- r_dtc_rx (using DTC transfer)
- r_dmaca_rx (using DMA transfer)

2.2.3 Operating Confirmation Environment

Table 2-1 shows the operating confirmation environment of this driver.

Table 2-1 Operation Confirmation Environment

Item	Contents
Integrated Development Environment	Renesas Electronics e^2 studio V5.20.020
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.05.00
	Compile Option：-lang = c99
Endian	Little Endian, Big Endian
USB Driver Revision Number	Rev.1.21
Using Board	Renesas Starter Kit for RX63N Renesas Starter Kit for RX64M Renesas Starter Kit for RX71M Renesas Starter Kit for RX65N
Host Environment	The operation of this USB Driver module connected to the following OSes has been confirmed. 1. Windows® 7 2. Windows® 8.1 3. Windows® 10

2.2.4 Usage of Interrupt Vector

Table 2-2 shows the interrupt vector which this driver uses.

Table 2-2 List of Usage Interrupt Vectors

Device	Contents
RX63N RX631	USBI0 Interrupt (Vector number: 35) / USBR0 Interrupt (Vector number: 90) USB D0FIFO0 Interrupt (Vector number: 33) / USB D1FIFO0 Interrupt (Vector number: 34)
	USBI1 Interrupt (Vector number: 38) / USBR1 Interrupt (Vector number: 91) USB D0FIFO1 Interrupt (Vector number: 36) / USB D1FIFO1 Interrupt (Vector number: 37)

RX64M RX71M	USBI0(GROUPB) Interrupt (Vector number: 189, Group interrupt source number： 62) USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35) USBR0 Interrupt (Vector number:90)
	USBAR Interrupt (Vector number: 94) USB D0FIFO2 Interrupt (Vector number: 32) / USB D1FIFO2 Interrupt (Vector number: 33)
RX65N RX651	USBI0(GROUPB) Interrupt (Vector number: 185, Group interrupt source number： 62) USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35) USBR0 Interrupt (Vector number:90)

2.2.5 Header Files

All API calls and their supporting interface definitions are located in r_usb_basic_if.h.

2.2.6 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*.

2.2.7 Compile Setting

For compile settings, refer to chapter **7, Configuration**.

2.2.8 ROM / RAM Size

The follows shows ROM/RAM size of this driver.

1. RX64M, RX71M, RX65N/RX651

	Checks arguments	Does not check arguments
ROM size	19.1K bytes (Note 2)	18.6K bytes (Note 3)
RAM size	9.1K bytes	9.1K bytes

2. RX63N/RX631

	Checks arguments	Does not check arguments
ROM size	16.8K bytes (Note 2)	16.3K bytes (Note 3)
RAM size	8.8K bytes	8.8K bytes

Note:

1. The default option is specified in the compiler optimization option.
2. The ROM size of “Checks arguments” is the value when *USB_CFG_ENABLE* is specified to *USB_CFG_PARAM_CHECKING* definition in *r_usb_basic_config.h* file.
3. The ROM size of “Does not check arguments” is the value when *USB_CFG_DISABLE* is specified to *USB_CFG_PARAM_CHECKING* definition in *r_usb_basic_config.h* file.

2.2.9 Argument

For the structure used in the argument of API function, refer to chapter **8, Structures**.

2.2.10 Adding the Module

This module must be added to an existing e^2 studio project. By using the e^2 studio plug-in, it is possible to update the include file path automatically. It is therefore recommended that this plug-in be used to add the project.

For instructions when using e^2 studio, refer to RX Family: Integration into e^2 studio, Firmware Integration Technology (document No. R01AN1723EU).

2.3 API (Application Programming Interface)

For the detail of the API function, refer to chapter **4, API Functions.**

2.4 Class Request

For the processing method when this driver receives the class request, refer to chapter **9, USB Class Requests**.

2.5 Descriptor

2.5.1 String Descriptor

This USB driver requires each string descriptor that is constructed to be registered in the string descriptor table. The following describes how to register a string descriptor.

1. First construct each string descriptor. Then, define the variable of each string descriptor in uint8_t* type.

Example descriptor construction)

```
uint8_t smp_str_descriptor0[] {
      0x04, /* Length */
      0x03, /* Descriptor type */
      0x09, 0x04 /* Language ID */
};
uint8_t smp_str_descriptor1[] =
{
      0x10, /* Length */
      0x03, /* Descriptor type */
      'R', 0x00,
      'E', 0x00,
      'N', 0x00,
      'E', 0x00,
      'S', 0x00,
      'A', 0x00,
      'S', 0x00
};
uint8_t smp_str_descriptor2[] =
{
      0x12, /* Length */
      0x03, /* Descriptor type */
      'C', 0x00,
      'D', 0x00,
      'C', 0x00,
      '_', 0x00,
      'D', 0x00,
      'E', 0x00,
      'M', 0x00,
      'O', 0x00
};
```

2. Set the top address of each string descriptor constructed above in the string descriptor table. Define the variables of the string descriptor table as uint8_t* type.

Note:

The position set for each string descriptor in the string descriptor table is determined by the index values set in the descriptor itself (iManufacturer, iConfiguration, etc.).
For example, in the table below, the manufacturer is described in smp_str_descriptor1 and the value of iManufacturer in the device descriptor is “1”. Therefore, the top address "smp_str_descriptor1" is set at Index “1” in the string descriptor table.

```
/* String Descriptor table */
uint8_t *smp_str_table[] =
{
      smp_str_descriptor0,    /* Index: 0 */
```

```
        smp_str_descriptor1,    /* Index: 1 */
        smp_str_descriptor2,    /* Index: 2 */
};
```

3. Set the top address of the string descriptor table in the *usb_descriptor_t* structure member (*string)*. Refer to chapter **8.4, usb_descriptor_t structure** for more details concerning the *usb_descriptor_t* structure.

2.5.2 Other Descriptors

1. Please construct the device descriptor, configuration descriptor, and qualifier descriptor based on instructions provided in the **Universal Serial Bus Revision 2.0 specification**(http://www.usb.org/developers/docs/) Each descriptor variable should be defined as uint8_t* type.
2. The top address of each descriptor should be registered in the corresponding *usb_descriptor_t* function member. For more details, refer to chapter **8.4, usb_descriptor_t structure.**

2.6 Peripheral Battery Charging (PBC)

This driver supports PBC.

PBC is the H/W control program for the target device that operates the Charging Port Detection (CPD) defined by the USB Battery Charging Specification (Revision 1.2).

You can get the result of CPD by calling *R_USB_GetInformation* function. For *R_USB_GetInformation* function, refer to chapter **4.11**.

Note:

RX63N/RX631 does not support PBC.

The processing flow of PBC is shown in Figure 2-1.

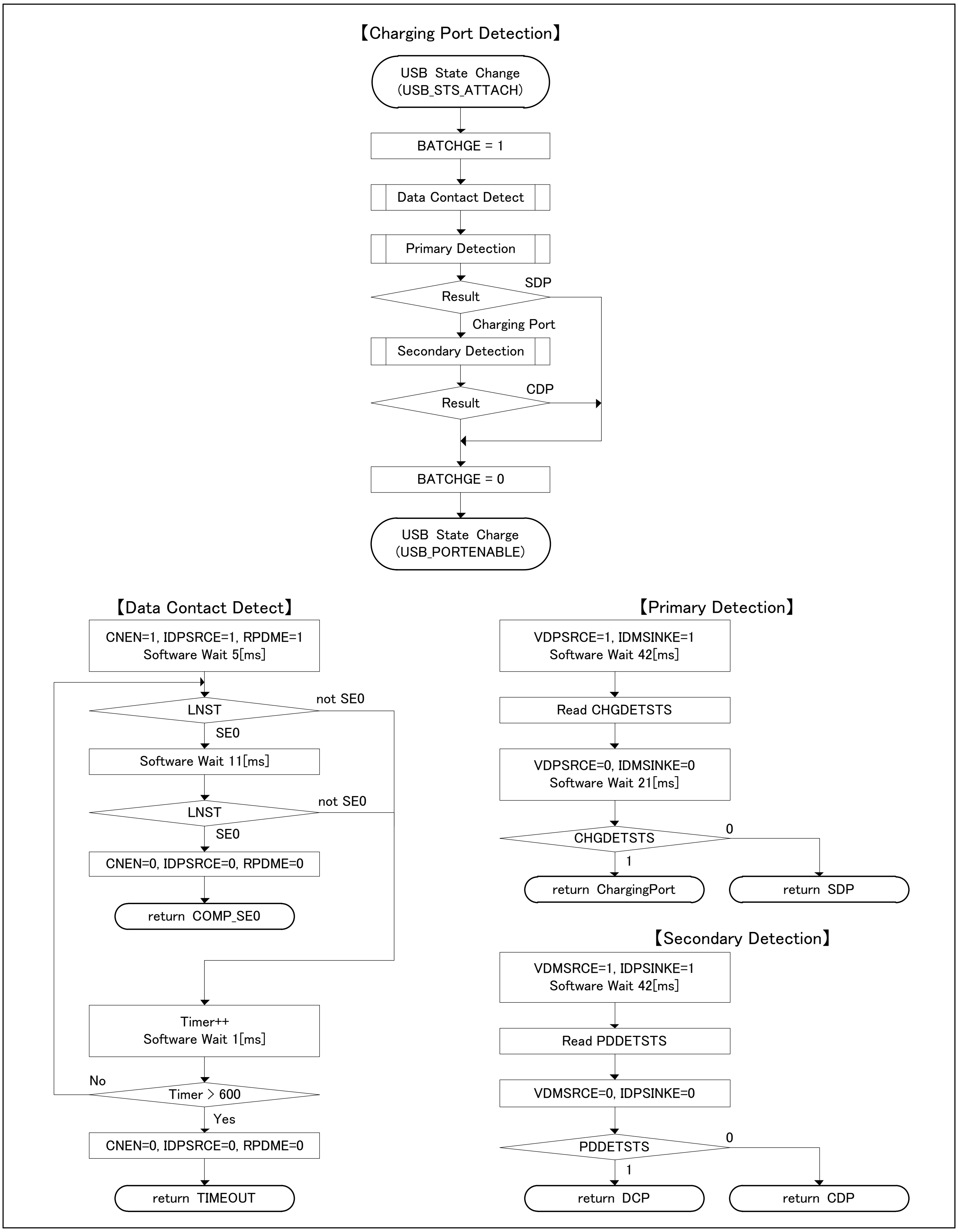

Figure 2-1 PBC processing flow

3. Host

3.1 Host Control Driver (HCD)

3.1.1 Basic function

HCD is a program for controlling the hardware. The functions of HCD are shown below.

1. Control transfer (Control Read, Control Write, No-data Control) and result notification.
2. Data transfer (bulk, interrupt) and result notification.
3. Data transfer suspension (all pipes).
4. USB communication error detection and automatic transfer retry
5. USB bus reset signal transmission and reset handshake result notification.
6. Suspend signal and resume signal transmission.
7. Attach/detach detection using ATCH and DTCH interrupts.

3.2 Host Manager (MGR)

3.2.1 Basic function

The functions of MGR are shown below.

1. Registration of HDCD.
2. State management for connected devices.
3. Enumeration of connected devices.
4. Searching for endpoint information from descriptors.

3.2.2 USB Standard Requests

MGR enumerates connected devices. The USB standard requests issued by MGR are listed below. The descriptor information obtained from a device is stored temporarily, and this information can be fetched by using the HCD API function.

GET_DESCRIPTOR（Device Descriptor）
SET_ADDRESS
GET_DESCRIPTOR（Configuration Descriptor）
SET_CONFIGURATION

3.3 API Information

This Driver API follows the Renesas API naming standards.

3.3.1 Hardware Requirements

This driver requires your MCU support the following features:

- USB

3.3.2 Software Requirements

This driver is dependent upon the following packages:

- r_bsp
- r_dtc_rx (using DTC transfer)
- r_dmaca_rx (using DMA transfer)

3.3.3 Operating Confirmation Environment

Table 3-1 shows the operating confirmation environment of this driver.

Table 3-1 Operation Confirmation Environment

Item	Contents
Integrated Development Environment	Renesas Electronics e^2 studio V5.20.020
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.05.00
	Compile Option：-lang = c99
Endian	Little Endian, Big Endian
USB Driver Revision Number	Rev.1.21
Using Board	Renesas Starter Kit for RX63N Renesas Starter Kit for RX64M Renesas Starter Kit for RX71M Renesas Starter Kit for RX65N

3.3.4 Usage of Interrupt Vector

Table 3-2 shows the interrupt vector which this driver uses.

Table 3-2 List of Usage Interrupt Vectors

Device	Contents
RX63N RX631	USBI0 Interrupt (Vector number: 35) / USBR0 Interrupt (Vector number: 90) USB D0FIFO0 Interrupt (Vector number: 33) / USB D1FIFO0 Interrupt (Vector number: 34)
RX64M RX71M	USBI0(GROUPB) Interrupt (Vector number: 189, Group interrupt source number：62) USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35) USBR0 Interrupt (Vector number:90)
	USBAR Interrupt (Vector number: 94) USB D0FIFO2 Interrupt (Vector number: 32) / USB D1FIFO2 Interrupt (Vector number: 33)
RX65N RX651	USBI0(GROUPB) Interrupt (Vector number: 185, Group interrupt source number：62) USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35) USBR0 Interrupt (Vector number:90)

3.3.5 Header Files

All API calls and their supporting interface definitions are located in *r_usb_basic_if.h*.

3.3.6 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*.

3.3.7 Compile Setting

For compile settings, refer to chapter **7, Configuration**..

3.3.8 ROM / RAM Size

The follows shows ROM/RAM size of this driver.

1. RX64M, RX71M, RX65N/RX651

	Checks arguments	Does not check arguments
ROM size	36.2K bytes (Note 2)	35.7K bytes (Note 3)
RAM size	16K bytes	16K bytes

2. RX63N/RX631

	Checks arguments	Does not check arguments
ROM size	33.4K bytes (Note 2)	32.9K bytes (Note 3)
RAM size	15.5K bytes	15.5K bytes

Note:

1. The default option is specified in the compiler optimization option.
2. The ROM size of “Checks arguments” is the value when *USB_CFG_ENABLE* is specified to *USB_CFG_PARAM_CHECKING* definition in *r_usb_basic_config.h* file.
3. The ROM size of “Does not check arguments” is the value when *USB_CFG_DISABLE* is specified to *USB_CFG_PARAM_CHECKING* definition in *r_usb_basic_config.h* file.

3.3.9 Argument

For the structure used in the argument of API function, refer to chapter **8, Structures**.

3.3.10 Adding the Module

This module must be added to an existing e^2 studio project. By using the e^2 studio plug-in, it is possible to update the include file path automatically. It is therefore recommended that this plug-in be used to add the project.

For instructions when using e^2 studio, refer to RX Family: Integration into e^2 studio, Firmware Integration Technology (document No. R01AN1723EU).

3.4 API (Application Programming Interface)

For the detail of the API function, refer to chapter **4, API Functions**.

3.5 Class Request

For the processing method when this driver receives the class request, refer to **9, USB Class Requests**.

3.6 How to Set the Target Peripheral List (TPL)

By registering the Vendor ID (VID) and Product ID (PID) in the USB host, USB communication will only be enabled for the USB device identified with a registered VID and PID.

To register a USB device in the TPL, specify the VID and PID as a set to the macro definitions listed in Table 3-3 in the configuration file (*r_usb_basic_config.h* file). The USB driver checks the TPL to make sure the VID and PID of the connected USB device are registered. If registration is confirmed, USB communication with the USB device is enabled. If the VID and PID are not registered in the TPL, USB communication is disabled.

If it is not necessary to register VID and PID in TPL, specify *USB_NOVENDOR* and *USB_NOPRODUCT* for the TPL definitions listed in Table 3-3. When *USB_NOVENDOR* and *USB_NOPRODUCT* are specified, the USB driver performs on TPL registration check, and this prevents situations from occurring in which USB communication is prevented because of the check.

Table 3-3 TPL Definition

Macro definition name	Description
USB_TPL_CNT	Specify the number of USB devices to be supported.
USB_TPL	Specify a VID/PID set for each USB device to be supported. (Always specify in the order of VID first, PID second.)
USB_HUB_TPL_CNT	Specify the number of USB hubs to be supported.
USB_HUB_TPL	Specify a VID/PID set for each USB hub to be supported. (Always specify in the order of VID first, PID second.)

== How to specify VID/PID in USB_TPL / USB_HUB_TP ==

#define USB_TPL $\underbrace{\underset{\text{VID}}{\text{0x0011}},\ \underset{\text{PID}}{\text{0x0022}}}_{\text{USB device 1}},\ \underbrace{\underset{\text{VID}}{\text{0x0033}},\ \underset{\text{PID}}{\text{0x0044}}}_{\text{USB device 2}},\ \underbrace{\underset{\text{VID}}{\text{0x0055}},\ \underset{\text{PID}}{\text{0x0066}}}_{\text{USB device 3}}$

#define USB_HUB_TPL $\underbrace{\underset{\text{VID}}{\text{0x1111}},\ \underset{\text{PID}}{\text{0x2222}}}_{\text{USB Hub1}},\ \underbrace{\underset{\text{VID}}{\text{0x3333}},\ \underset{\text{PID}}{\text{0x4444}}}_{\text{USB Hub2}}$

Example 1) Register 3 USB devices and 2 USB hubs in the TPL

```
#define     USB_CFG_TPLCNT          3
#define     USB_CFG_TPL             0x0011, 0x0022, 0x0033, 0x0044, 0x0055, 0x0066
#define     USB_CFG_HUB_TPLCNT      2
#define     USB_HUB_TPL             0x1111, 0x2222, 0x3333, 0x4444
```

Example 2) Register 3 USB devices (no USB hubs) in the TPL

```
#define     USB_CFG_TPLCNT          3
#define     USB_CFG_TPL             0x0011, 0x0022, 0x0033, 0x0044, 0x0055, 0x0066
#define     USB_CFG_HUB_TPLCNT      1
#define     USB_CFG_HUB_TPL         USB_NOVENDOR,USB_NOPRODUCT
```

Example 3) VID and PID registration not required

```
#define     USB_CFG_TPLCNT          1
#define     USB_CFG_TPL             USB_NOVENDOR,USB_NOPRODUCT
#define     USB_CFG_HUB_TPLCNT      1
```

```
#define    USB_CFG_HUB_TPL    USB_NOVENDOR,USB_NOPRODUCT
```

Note:

1. Set *USB_CFG_TPLCNT* and *USB_CFG_HUB_TPLCNT* to 1, even if *USB_NOVENDOR* and *USB_NOPRODUCT* are specified for the TPL definitions in Table 3-3.
2. For the configuration file (*r_usb_basic_config.h*), refer to chapter **7.**

3.7 Allocation of Device Addresses

In USB Host mode, the USB driver allocates device addresses to the connected USB devices.

1. When a USB Hub is used

   Device address value 1 is allocated to a USB Hub, and device address values 2 and thereafter are allocated to USB devices connected to the Hub.

2. When a USB Hub is not used

   Device address value 1 is allocated to the USB device.

Note:

Device addresses are allocated in USB module units. For example, in the case of an MCU that supports multiple USBs such as RX64M, if both the USB0 module and the USB1 module are connected to USB devices, device address value 1 is allocated to each USB device.

3.8 Host Battery Charging (HBC)

This driver supports HBC.
HBC is the H/W control program for the target device that operates the CDP or the DCP as defined by the USB Battery Charging Specification Revision 1.2.
Processing is executed as follows according to the timing of this driver. Refer to Figure 3-1.

VBUS is driven
Attach processing
Detach processing

Moreover, processing is executed in coordination with the PDDETINT interrupt.
There is no necessity for control from the upper layer.

You can get the result of Change Port Detection (CPD) by calling *R_USB_GetInformation* function. For *R_USB_GetInformation* function, refer to chapter .

Note:

RX63N/RX631 does not support HBC.

The processing flow of HBC is shown Figure 3-1.

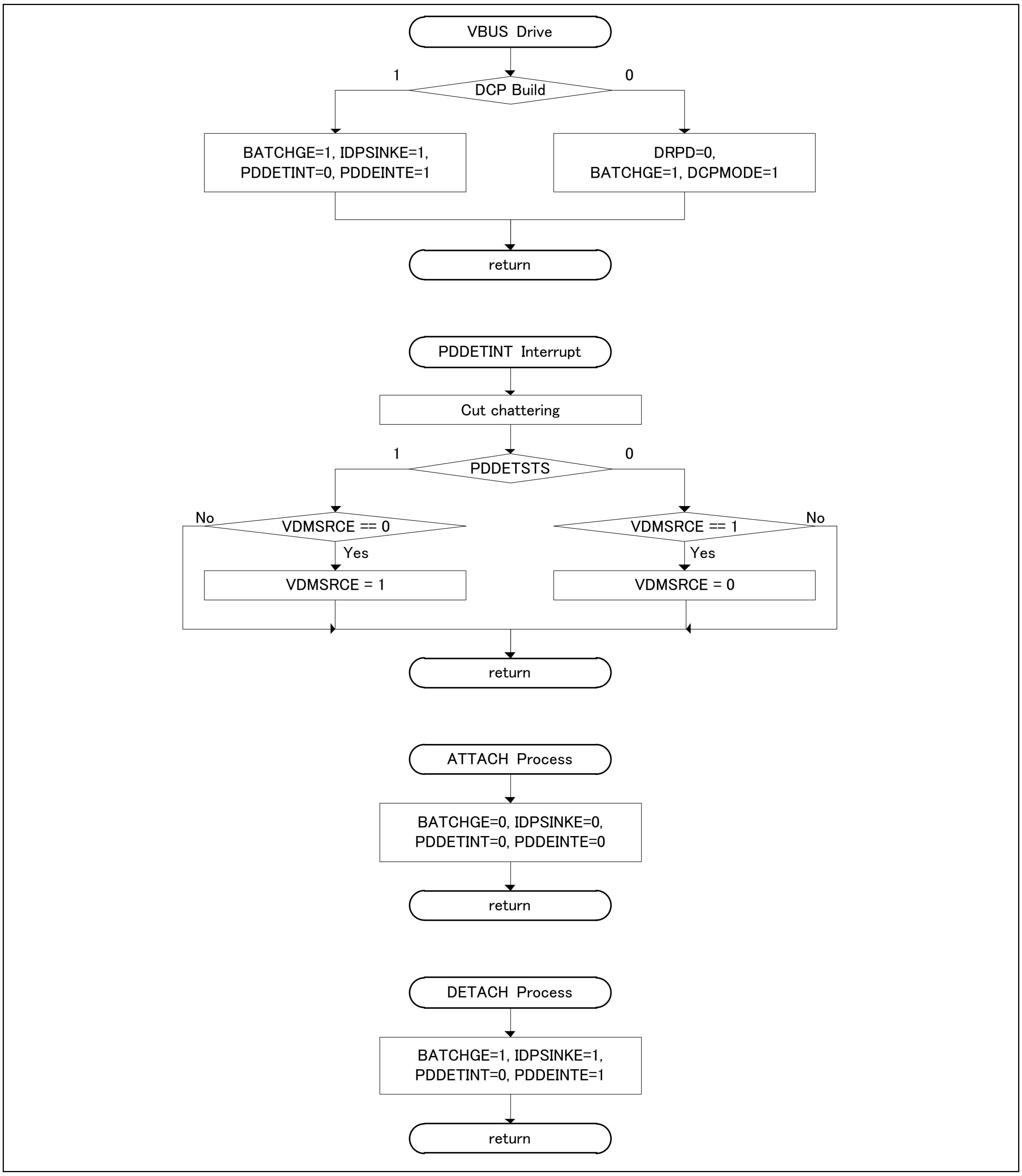

Figure 3-1 HBC processing flow

4. API Functions

Table 4-1 provides a list of API functions. These APIs can be used in common for all the classes. Use the APIs below in application programs.

Table 4-1 List of API Functions

API	Description
R_USB_Open()	Start the USB module
R_USB_Close()	Stop the USB module
R_USB_GetVersion()	Get the driver version
R_USB_Read()	Request USB data read
R_USB_Write()	Request USB data write
R_USB_Stop()	Stop USB data read/write processing
R_USB_Suspend()	Request suspend
R_USB_Resume()	Request resume
R_USB_GetEvent()	Return USB-related completed events
R_USB_VbusSetting()	Sets VBUS supply start/stop.
R_USB_GetInformation()	Get information on USB device.
R_USB_PipeRead()	Request data read from specified pipe
R_USB_PipeWrite()	Request data write to specified pipe
R_USB_PipeStop()	Stop USB data read/write processing to specified pipe
R_USB_GetUsePipe()	Get pipe number
R_USB_GetPipeInfo()	Get pipe information

Note:

1. The class-specific API function other than the above API is supported in Host Mass Storage Class. Refer to the document (Document number: R01AN2026) for the class-specific API.
2. The class-specific API function other than the above API is supported in Host Human Interface Device Class. Refer to the document (Document number: R01AN2028) for the class-specific API.

4.1 R_USB_Open

Power on the USB module and initialize the USB driver. (This is a function to be used first when using the USB module.)

Format

usb_err_t	R_USB_Open(usb_ctrl_t *p_ctrl, usb_cfg_t *p_cfg)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
p_cfg	Pointer to usb_cfg_t structure area

Return Value

USB_SUCCESS	Success
USB_ERR_PARA	Parameter error
USB_ERR_BUSY	Specified USB module now in use

Description

This function applies power to the USB module specified in the argument (*p_ctrl*).

Note

1. For details concerning the *usb_ctrl_t* structure, see chapter **8.1, usb_ctrl_t structure**, and for the *usb_cfg_t* structure, see chapter **8.2, usb_setup_t structure**.
2. Specify the number of the module (*USB_IP0/USB_IP1*) to be started up in member (*module)* of the *usb_ctrl_t* structure. Specify “*USB_IP0*” to start up the USB0 module and “*USB_IP1*” to start up the USB1 module. If something other than *USB_IP0* or *USB_IP1* is assigned to the member (*module*), then *USB_ERR_PARA* will be the return value.
3. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to t the member (*module*). If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
4. In the *usb_cfg_t* structure member (*usb_mode)*, specify “*USB_HOST*” to start up USB host operations and “*USB_PERI*” to start up USB peripheral operations If these settings are not supported by the USB module, *USB_ERR_PARA* will be returned.
5. In USB Peripheral mode, Specify the USB speed (*USB_HS / USB_FS*) in the *usb_ctrl_t* structure member (*usb_speed)*. If the speed set in the member is not supported by the USB module, *USB_ERR_PARA* will be returned. In USB Host mode, it is not necessary to set the USB speed to the member (*usb_speed*).
6. Assign a pointer to the usb_descriptor_t structure to the member (*usb_reg*) of the *usb_cfg_t* structure. This assignment is only effective if “*USB_PERI*” is assigned to the member (*usb_mode*). If “*USB_HOST*” is assigned, then assignment to the member (*usb_reg*) is ignored.

Examples

1. **In the case of USB Host mode**

```
void usb_host_application(void)
{
        usb_err_t       err;
        usb_ctrl_t      ctrl;
        usb_cfg_t       cfg;
                        :
        ctrl.module = USB_IP0;
        cfg.usb_mode = USB_HOST;
        cfg.usb_speed = USB_HS;
        err = R_USB_Open(&ctrl, &cfg);   /* Start USB module */
        if (USB_SUCCESS != err)
        {
                        :
        }
                        :
}
```

2. **In the case of USB Peripheral**

```
usb_descriptor_t smp_descriptor =
{
        g_device,
        g_config_f,
        g_config_h,
        g_qualifier,
        g_string
};
void    usb_peri_application(void)
{
        usb_err_t   err;
        usb_ctrl_t  ctrl;
        usb_cfg_t   cfg;
                        :
        ctrl.module = USB_IP1;
        cfg.usb_mode = USB_PERI;
        cfg.usb_speed = USB_HS;
        cfg.usb_reg = &smp_descriptor;
        err = R_USB_Open(&ctrl, &cfg );   /* Start USB module */
        if (USB_SUCCESS != err)
        {
                        :
        }
                        :
}
```

4.2 R_USB_Close

Power off USB module.

Format

usb_err_t	R_USB_Close(usb_ctrl_t *p_ctrl)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area

Return Value

USB_SUCCESS	Success
USB_ERR_PARA	Parameter error
USB_ERR_NOT_OPEN	USB module is not open.
USB_ERR_NG	USB module close processing failed.

Description

This function terminates power to the USB module specified in argument (*p_ctrl*). USB0 module stops when *USB_IP0* is specified to the member (*module)*, USB1 module stops when *USB_IP1* is specified to the member (*module).*

Note

1. Specify the number of the USB module (*USB_IP0/USB_IP1*) to be stopped in the *usb_ctrl_t* structure member (*module)*. If something other than *USB_IP0* or *USB_IP1* is assigned to the member (*module)*, then *USB_ERR_PARA* will be the return value.
2. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member (*module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.

Example

```
void    usr_application(void)
{
  usb_err_t     err;
  usb_ctrl_t    ctrl;
                :
  ctrl.module = USB_IP0
  err = R_USB_Close(&ctrl);
  if (USB_SUCCESS != err)
  {
                :
  }
                :
}
```

4.3 R_USB_GetVersion

Return API version number

Format

```
usb_err_t              R_USB_GetVersion()
```

Arguments

—	—

Return Value

Version number

Description

The version number of the USB driver is returned.

Note

--

Example

```
void     usr_application( void )
{
    uint32_t     version;
              :
    version = R_USB_GetVersion();
              :
}
```

4.4 R_USB_Read

USB data read request

Format

usb_err_t	R_USB_Read(usb_ctrl_t *p_ctrl, uint8_t *p_buf, uint32_t size)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
p_buf	Pointer to area that stores read data
size	Read request size

Return Value

USB_SUCCESS	Successfully completed (Data read request completed)
USB_ERR_PARA	Parameter error
USB_ERR_BUSY	Data receive request already in process for USB device with same device address.
USB_ERR_NG	Other error

Description

1. Bulk/interrupt data transfer

   Requests USB data read (bulk/interrupt transfer).
   The read data is stored in the area specified by argument (*p_buf*).
   After data read is completed, confirm the operation by checking the return value (*USB_STS_READ_COMPLETE*) of the *R_USB_GetEvent* function. The received data size is set in member *(size)* of the *usb_ctrl_t* structure. To figure out the size of the data when a read is complete, check the return value (*USB_STS_READ_COMPLETE*) of the *R_USB_GetEvent* function, and then refer to the member *(size)* of the *usb_crtl_t* structure.

2. Control data transfer

   Refer to chapter **9, USB Class Requests** for details.

Note

1. This API only performs data read request processing. An application program does not wait for data read completion by using this API.
2. When *USB_SUCCESS* is returned for the return value, it only means that a data read request was performed to the USB driver, not that the data read processing has completed. The completion of the data read can be checked by reading the return value (*USB_STS_READ_COMPLETE*) of the *R_USB_GetEvent* function.
3. When the read data is n times the maximum packet size and does not meet the read request size, the USB driver assumes the data transfer is still in process and *USB_STS_READ_COMPLETE* is not set as the return value of the *R_USB_GetEvent* function.
4. Before calling this API, assign the device class type (see chapter **6, Device Class Types**) to the member *(type)* of the *usb_ctrl_t* structure. In USB Host mode, in order to identify the USB device to be accessed, assign the USB module number (*USB_IP0* or *USB_IP1*) to the member *(module)*, and assign the device address to the member *(address)*. If something other than *USB_IP0* or *USB_IP1* is assigned to the member *(module)* or if an unsupported device class type is assigned to the member *(type)*, then *USB_ERR_PARA* will be the return value.
5. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
6. Do not assign a pointer to the auto variable (stack) area to the second argument (*p_buf*).
7. The size of area assigned to the second argument (*p_buf*) must be at least as large as the size specified for the third argument (*size*).
8. If 0 (zero) is assigned to one of the arguments, *USB_ERR_PARA* will be the return value.
9. In USB Host mode it is not possible to repeatedly call the *R_USB_Read* function with the same values assigned to the member *(type)* and the member *(address)* of the *usb_crtl_t* structure. If the *R_USB_Read*

function is called repeatedly, then *USB_ERR_BUSY* will be the return value. To call the *R_USB_Read* function more than once with the same values assigned to the members (*type* and *address*), first check the *USB_STS_READ_COMPLETE* return value from the *R_USB_GetEvent* function, and then call the *R_USB_Read* function.

10. In USB Peripheral mode it is not possible to repeatedly call the *R_USB_Read* function with the same value assigned to the member *(type)* of the *usb_crtl_t* structure. If the *R_USB_Read* function is called repeatedly, then *USB_ERR_BUSY* will be the return value. To call the *R_USB_Read* function more than once with the same value assigned to the member *(type)* , first check the *USB_STS_READ_COMPLETE* return value from the *R_USB_GetEvent* function, and then call the *R_USB_Read* function.
11. In Vendor Class, use the *R_USB_PipeRead* function. Do not assign *USB_HVND/USB_PVND* to the member *(type)* of the *usb_ctrl_t* structure. If *USB_HVND/USB_PVND* is assigned, then *USB_ERR_PARA* will be the return value.
12. If this API is called after assigning *USB_PCDCC*, *USB_HMSC*, or *USB_PMSC* to the member *(type)* of the *usb_crtl_t* structure, then *USB_ERR_PARA* will be the return value.
13. In Host Mass Storage Class, to access storage media, use the FAT (File Allocation Table) API rather than this API. If this API is used, then *USB_ERR_NG* will be the return value.
14. In the USB device is in the CONFIGURED state, this API can be called. If this API is called when the USB device is in other than the CONFIGURED state, then *USB_ERR_NG* will be the return value.

Example

```
void usb_application( void )
{
    usb_ctrl_t      ctrl;
                        :
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                        :
            case USB_STS_WRITE_COMPLETE:
                        :
                ctrl.module = USB_IP1
                ctrl.adderss = adr;
                ctrl.type = USB_HCDC;
                R_USB_Read(&ctrl, g_buf, DATA_LEN);
                        :
            break;
            case USB_STS_READ_COMPLETE:
                        :
            break;
                        :
        }
    }
}
```

4.5 R_USB_Write

USB data write request

Format

usb_err_t	R_USB_Write(usb_ctrl_t *p_ctrl, uint8_t *p_buf, uint32_t size)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
p_buf	Pointer to area that stores write data
size	Write size

Return Value

USB_SUCCESS	Successfully completed (Data write request completed)
USB_ERR_PARA	Parameter error
USB_ERR_BUSY	Data write request already in process for USB device with same device address.
USB_ERR_NG	Other error

Description

1. Bulk/Interrupt data transfer

   Requests USB data write (bulk/interrupt transfer).
   Stores write data in area specified by argument (*p_buf*).
   Set the device class type in *usb_ctrl_t* structure member (*type).*
   Confirm after data write is completed by checking the return value (*USB_STS_WRITE_COMPLETE*) of the *R_USB_GetEvent* function.
   To request the transmission of a NULL packet, assign *USB_NULL*(0) to the third argument (*size*).

2. Control data transfer

   Refer to chapter **9, USB Class Requests** for details.

Note

1. This API only performs data write request processing. An application program does not wait for data write completion by using this API.
2. When *USB_SUCCESS* is returned for the return value, it only means that a data write request was performed to the USB driver, not that the data write processing has completed. The completion of the data write can be checked by reading the return value (*USB_STS_WRITE_COMPLETE)* of the *R_USB_GetEvent* function.
3. Before calling this API, assign the device class type (see chapter 6, Device Class Types) to the member *(type)* of the *usb_ctrl_t* structure. In USB Host mode, in order to identify the USB device to be accessed, assign the USB module number (*USB_IP0* or *USB_IP1*) to the member *(module)*, and assign the device address to the member *(address)*. If something other than *USB_IP0* or *USB_IP1* is assigned to the member *(module)* or if an unsupported device class type is assigned to the member *(type)*, then *USB_ERR_PARA* will be the return value.
4. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
5. Do not assign a pointer to the auto variable (stack) area to the second argument (*p_buf*).
6. The size of area assigned to the second argument (*p_buf*) must be at least as large as the size specified for the third argument (*size*).
7. If *USB_NULL* is assigned to the argument (*p_ctrl*), then *USB_ERR_PARA* will be the return value.
8. If a value other than 0 (zero) is set for the argument (*size*) and *USB_NULL* is assigned to the argument (*p_buf*), then *USB_ERR_PARA* will be the return value.
9. In USB Host mode it is not possible to repeatedly call the *R_USB_Write* function with the same values assigned to the member *(type)* and the member *(address)* of the *usb_crtl_t* structure. If the *R_USB_Write* function is called repeatedly, then *USB_ERR_BUSY* will be the return value. To call the *R_USB_Write* function more than once with the same values assigned to the members (*type* and *address*), first check the

USB_STS_WRITE_COMPLETE return value from the *R_USB_GetEvent* function, and then call the *R_USB_Write* function.

10. In USB Peripheral mode it is not possible to repeatedly call the *R_USB_Write* function with the same value assigned to the member *(type)* of the *usb_crtl_t* structure. If the *R_USB_Write* function is called repeatedly, then *USB_ERR_BUSY* will be the return value. To call the *R_USB_Write* function more than once with the same value assigned to the member *(type)*, first check the *USB_STS_WRITE_COMPLETE* return value from the *R_USB_GetEvent* function, and then call the *R_USB_Write* function.
11. In Vendor Class, use the *R_USB_PipeWrite* function. Do not assign *USB_HVND/USB_PVND* to the member *(type)* of the *usb_ctrl_t* structure. If *USB_HVND/USB_PVND* is assigned, then *USB_ERR_PARA* will be the return value.
12. If this API is called after assigning *USB_HCDCC*, *USB_HMSC*, or *USB_PMSC* to the member *(type)* of the *usb_crtl_t* structure, then *USB_ERR_PARA* will be the return value.
13. In Host Mass Storage Class, to access storage media, use the FAT (File Allocation Table) API rather than this API. If this API is used, then *USB_ERR_NG* will be the return value.
14. This API can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

```
void usb_application( void )
{
    usb_ctrl_t      ctrl;
                    :
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                        :
            case USB_STS_READ_COMPLETE:
                        :
                ctrl.module = USB_IP0;
                ctrl.address = adr;
                ctrl.type = USB_HCDC;
                R_USB_Write(&ctrl, g_buf, size);
                        :
            break;
            case USB_STS_WRITE_COMPLETE:
                        :
            break;
                        :
        }
    }
}
```

4.6 R_USB_Stop

USB data read/write stop request

Format

usb_err_t	R_USB_Stop(usb_ctrl_t *p_ctrl, uint16_t type)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
type	Receive (USB_READ) or send (USB_WRITE)

Return Value

USB_SUCCESS	Successfully completed (stop completed)
USB_ERR_PARA	Parameter error
USB_ERR_NG	Other error

Description

This function is used to request a data read/write transfer be terminated when a data read/write transfer is performing.
To stop a data read, set *USB_READ* as the argument (*type*); to stop a data write, specify *USB_WRITE* as the argument (*type)*.

Note

1. Before calling this API, assign the device class type to the member *(type)* of the *usb_ctrl_t* structure. In USB Host mode, in order to identify the USB device to be accessed, assign the USB module number (*USB_IP0* or *USB_IP1*) to the member *(module)*, and assign the device address to the the member *(address)*. If something other than *USB_IP0* or *USB_IP1* is assigned to the member *(module)* or if an unsupported device class type is assigned to the member *(type)*, then *USB_ERR_PARA* will be the return value.
2. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
3. If something other than *USB_READ* or *USB_WRITE* is assigned to the third argument (*type*), then *USB_ERR_PARA* will be the return value.
4. In USB Host mode, *USB_ERR_NG* will be the return value when this API can not stop the data read/write request.
5. If this API is called after assigning *USB_HMSC* or *USB_PMSC* to the member *(type)* of the *usb_crtl_t* structure, then *USB_ERR_PARA* will be the return value.
6. In Vendor Class, use the *R_USB_PipeStop* function. If this API is used for Vender Class, then *USB_ERR_NG* will be the return value.
7. Do not use this API for the Host Mass Storage Class. If this API is used, *USB_ERR_NG* will be the return value.
8. This API can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

```
void usb_application( void )
{
    usb_ctrl_t      ctrl;
                        :
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                        :
            case USB_STS_DETACH:
                        :
                ctrl.module = USB_IP1;
                ctrl.address = adr;
                ctrl.type = USB_HCDC;
                R_USB_Stop(&ctrl, USB_READ );     /* Receive stop */
                R_USB_Stop(&ctrl, USB_WRITE );    /* Send stop */
                        :
            break;
                        :
        }
    }
}
```

4.7 R_USB_Suspend

Suspend signal transmission

Format

usb_err_t	R_USB_Suspend(usb_ctrl_t *p_ctrl)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area

Return Value

USB_SUCCESS	Successfully completed
USB_ERR_PARA	Parameter error
USB_ERR_BUSY	There is a suspend request or a suspend condition on the USB device with the same device address.
USB_ERR_NG	Other error

Description

Sends a SUSPEND signal from the USB module assigned to the member *(module)* of the *usb_crtl_t* structure.

Note

1. This API only performs a Suspend signal transmission. An application program does not wait for Suspend signal transmission completion by using this API.
2. This API can only be used in USB host mode. If this API is used in USB Peripheral mode, then *USB_ERR_NG* will be the return value.
3. This API does not support the Selective Suspend function.
4. Assign the USB module to which a SUSPEND signal is transmitted to the member *(module)* of the *usb_ctrl_t* structure. *USB_IP0* or *USB_IP1* should be assigned to the member *(module)*. If something other than *USB_IP0* or *USB_IP1* is assigned to the member *(module)* or if an unsupported device class type is assigned to the member *(type)*, then *USB_ERR_PARA* will be the return value.
5. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
6. This API can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

```
void    usb_host_application( void )
{
        usb_ctrl_t ctrl;
                                :
        while (1)
        {
                switch (R_USB_GetEvent(&ctrl))
                {
                                        :
                        case USB_STS_NONE:
                                        :
                                ctrl.module = USB_IP0;
                                R_USB_Suspend(&ctrl);
                        break;
                        case USB_STS_SUSPEND:
                                        :
                        break;
                                        :
                  }
          }
}
```

4.8 R_USB_Resume

Resume signal transmission

Format

usb_err_t	R_USB_Resume(usb_ctrl_t *p_ctrl)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area

Return Value

USB_SUCCESS	Successfully completed
USB_ERR_PARA	Parameter error
USB_ERR_BUSY	Resume already requested for same device address (USB host mode only)
USB_ERR_NOT_SUSPEND	USB device is not in the SUSPEND state.
USB_ERR_NG	Other error

Description

This function sends a RESUME signal from the USB module assigned to the member *(module)* of the *usb_ctrl_t* structure.

After the resume request is completed, confirm the operation with the return value (*USB_STS_RESUME*) of the *R_USB_GetEvent* function

Note

1. This API only performs a Resume signal transmission request. An application program does not wait for Resume signal transmission completion by using this API.
2. Call this API after calling the *R_USB_Open* function (and before calling the *R_USB_Close* function).
3. This API can be used for RemoteWakeup only with HID Class in USB Peripheral mode. In this case, the USB module number is not required to be assigned to the member *(module)* of the *usb_ctrl_t* structure.
4. Assign the USB module to which the RESUME signal is transmitted to the member *(module)* of the *usb_ctrl_t* structure. *USB_IP0* or *USB_IP1* should be assigned to the member *(module)*. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
5. This API can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

1. **In the case of USB Host mode**

```
void     usb_host_application( void )
{
        usb_ctrl_t ctrl;
                            :
        while (1)
        {
                switch (R_USB_GetEvent( &ctrl ))
                {
                            :
                        case USB_STS_NONE:
                            :
                                ctrl.module = USB_IP0;
                                R_USB_Resume( &ctrl );
                            :
                        break;
                        case USB_STS_RESUME:
                            :
                        break;
                            :
                  }
        }
}
```

2. **In the case of HID device(USB Peripheral)**

```
void     usb_peri_application( void )
{
        usb_ctrl_t ctrl;
                            :
        while (1)
        {
                switch (R_USB_GetEvent( &ctrl ))
                {
                            :
                        case USB_STS_NONE:
                            :
                                R_USB_Resume(&ctrl);
                            :
                        break;
                        case USB_STS_RESUME:
                            :
                        break;
                            :
                  }
        }
}
```

4.9 R_USB_GetEvent

Get completed USB-related events

Format

usb_err_t	R_USB_GetEvent(usb_ctrl_t *p_ctrl)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area

Return Value

--	Value of completed USB-related events

Description

This function obtains completed USB-related events.

In USB host mode, the device address value of the USB device that completed an event is specified in the *usb_ctrl_t* structure member *(address)* specified by the event's argument. In USB peripheral mode, *USB_NULL* is specified in member *(address)*.

Note

1. Call this API after calling the *R_USB_Open* function (and before calling the *R_USB_Close* function).
2. Refer to chapter **5, Return Value of R_USB_GetEvent Function**" for details on the completed event value used as the API return value.
3. If there is no completed event when calling this API, then *USB_STS_NONE* will be the return value.

Example

```
void usb_host_application( void )
{
    usb_ctrl_t      ctrl;
                              :
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                              :
            case USB_STS_CONFIGURED:
                              :
            break;
                              :
        }
    }
}
```

4.10 R_USB_VbusSetting

VBUS Supply Start/Stop Specification

Format

usb_err_t	R_USB_VbusSetting(usb_ctrl_t *p_ctrl, uint16_t state)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
state	VBUS supply start/stop specification

Return Value

USB_SUCCESS	Successful completion (VBUS supply start/stop completed)
USB_ERR_PARA	Parameter error
USB_ERR_NG	Other error

Description

Specifies starting or stopping the VBUS supply.

Note

1. For information on setting the VBUS output of the power source IC for the USB Host to either Low Assert or High Assert, see the setting of the *USB_CFG_VBUS* definition described in chapter **7.1.1, r_usb_basic_config.h.**
2. Assign the module number *(USB_IP0/USB_IP1*) to specify starting or stopping the VBUS supply to the member *(module)* of the first argument (*p_ctrl*). If ”*USB_IP0*” is assigned, setting is applied to the USB0 module. If ”*USB_IP1*” is assigned, setting is applied to the USB1 module. If something other than *USB_IP0* or *USB_IP1* is assigned to the member *(module)*, then *USB_ERR_PARA* will be the return value.
3. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
4. Assign "*USB_ON*" or "*USB_OFF*" to the second argument. Assign "*USB_ON*" in order to start the VBUS supply, and assign "*USB_OFF*" in order to stop the VBUS supply.
5. This API is processed only in USB Host mode. If this API is called in USB Peripheral mode, then *USB_ERR_NG* will be the return value.

Example

```
void usb_host_application( void )
{
    usb_ctrl_t ctrl;
             :
             :
    ctrl.module = USB_IP0;
    R_USB_VbusSetting( &ctrl, USB_ON ); /* Start VBUS supply */
             :
             :
    ctrl.module = USB_IP0;
    R_USB_VbusSetting( &ctrl, USB_OFF ); /* Stop VBUS supply */
             :
             :
}
```

4.11 R_USB_GetInformation

Get USB device information

Format

usb_err_t	R_USB_GetInformation(usb_ctrl_t *p_ctrl, usb_info_t *p_info)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
p_info	Pointer to usb_info_t structure area

Return Value

USB_SUCCESS	Successful completion (VBUS supply start/stop completed)
USB_ERR_PARA	Parameter error

Description

This function obtains completed USB-related events.

For information to be obtained, see chpater **8.6, usb_info_t structure**.

Note

1. Call this API after calling the *R_USB_Open* function (and before calling the *R_USB_Close* function).
2. In USB Host mode, in order to identify the USB device to obtain information, assign the USB module number (*USB_IP0/USB_IP1*) to the member *(module)*, and assign the device address to the member *(address)*. If something other than USB_IP0 or USB_IP1 is assigned to the member *(module)*, then *USB_ERR_PARA* will be the return value.
3. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
4. In USB Peripheral mode, assign *USB_NULL* to the first arugument (*p_ctrl*).

Example

1. In the case of USB Host mode

```
void usb_host_application( void )
{
        usb_ctrl_t ctrl;
        usb_info_t info;
                                :
        ctrl.module = USB_IP0;
        ctrl.address = adr;
        R_USB_GetInformation( &ctrl, &info );
                                :
}
```

2. In the case of USB Peripheral mode

```
void usb_peri_application( void )
{
        usb_ctrl_t ctrl;
        usb_info_t info;
                        :
        R_USB_GetInformation( (usb_ctrl_t *)USB_NULL, &info );
                        :
}
```

4.12 R_USB_PipeRead

Request data read via specified pipe

Format

usb_err_t	R_USB_PipeRead(usb_ctrl_t *p_ctrl, uint8_t *p_buf, uint32_t size)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
p_buf	Pointer to area that stores data
size	Read request size

Return Value

USB_SUCCESS	Successfully completed
USB_ERR_PARA	Parameter error
USB_ERR_BUSY	Specifed pipe now handling data receive/send request
USB_ERR_NG	Other error

Description

This function requests a data read (bulk/interrupt transfer) via the pipe specified in the argument.

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

After the data read is completed, confirm the operation with the *R_USB_GetEvent* function return value (*USB_STS_READ_COMPLETE*). To figure out the size of the data when a read is complete, check the return value (*USB_STS_READ_COMPLETE*) of the R_USB_GetEvent function, and then refer to the member (*size*) of the *usb_crtl_t* structure.

Note

1. This API only performs data read request processing. An application program does not wait for data read completion by using this API.
2. When *USB_SUCCESS* is returned for the return value, it only means that a data read request was performed to the USB driver, not that the data read processing has completed. The completion of the data read can be checked by reading the return value (*USB_STS_READ_COMPLETE*) of the *R_USB_GetEvent* function.
3. When the read data is n times the max packet size and does not meet the read request size, the USB driver assumes the data transfer is still in process and *USB_STS_READ_COMPLETE* is not set as the return value of the *R_USB_GetEvent* function.
4. Before calling this API, assign the PIPE number (*USB_PIPE1* to *USB_PIPE9*) to be used to the member *(pipe)* of the usb_ctrl_t structure. In USB Host mode, in order to identify the USB device to be accessed, assign the USB module number (*USB_IP0* or *USB_IP1*) to the member *(module)*, and assign the device address to the member *(address)*. If something other than *USB_IP0* or *USB_IP1* is assigned to the member *(module)*, then *USB_ERR_PARA* will be the return value.
5. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
6. If something other than *USB_PIPE1* through *USB_PIPE9* is assigned to the member *(pipe)* of the *usb_ctrl_t* structure, then *USB_ERR_PARA* will be the return value.
7. Do not assign a pointer to the auto variable (stack) area to the second argument (*p_buf*).
8. The size of area assigned to the second argument (*p_buf*) must be at least as large as the size specified for the third argument *(size)*.
9. If 0 (zero) is assigned to one of the arguments, then *USB_ERR_PARA* will be the return value.
10. It is not possible to repeatedly call the *R_USB_PipeRead* function with the same value assigned to the member *(pipe)* of the usb_crtl_t structure. If the *R_USB_PipeRead* function is called repeatedly, then *USB_ERR_BUSY* will be the return value. To call the *R_USB_PipeRead* function more than once with the same value assigned

to the member *(pipe)*, first check the *USB_STS_READ_COMPLETE* return value from the *R_USB_GetEvent* function, and then call the *R_USB_PipeRead* function.

11. In CDC/HID Class, to perform a Bulk/Interrupt transfer, use the *R_USB_Read* function rather than this API. With Host Mass Storage Class, to perform data access to the MSC device, use the FAT (File Allocation Table) API rather than this API.
12. Assign nothing to the member *(type)* of the *usb_ ctrl_t* structure. Even if the device class type or something is assigned to the member *(type)*, it is ignored.
13. To transfer the data for a Control transfer, use the *R_USB_Read* function rather than this API. If this API is used, then *USB_ERR_NG* will be the return value.
14. This API can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

```
void usb_application( void )
{
    usb_ctrl_t ctrl;
                        :
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                        :
            case USB_STS_WRITE_COMPLETE:
                        :
                ctrl.module = USB_IP1;
                ctrl.pipe = USB_PIPE1;
                R_USB_PipeRead(&ctrl, buf, size);
                        :
            break;
            case USB_STS_READ_COMPLETE:
                        :
            break;
                        :
        }
    }
}
```

4.13 R_USB_PipeWrite

Request data write to specified pipe

Format

usb_err_t	R_USB_PipeWrite(usb_ctrl_t *p_ctrl, uint8_t *p_buf, uint32_t size)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
p_buf	Pointer to area that stores data
size	Write request size

Return Value

USB_SUCCESS	Successfully completed
USB_ERR_PARA	Parameter error
USB_ERR_BUSY	Specifed pipe now handling data receive/send request
USB_ERR_NG	Other error

Description

This function requests a data write (bulk/interrupt transfer).

The write data is stored in the area specified in the argument (*p_buf*).

After data write is completed, confirm the operation with the return value (*USB_STS_WRITE_COMPLETE*) of the *R_USB_GetEvent* function.

To request the transmission of a NULL packet, assign *USB_NULL* (0) to the third argument (*size*).

Note

1. This API only performs data write request processing. An application program does not wait for data write completion by using this API.
2. When *USB_SUCCESS* is returned for the return value, it only means that a data write request was performed to the USB driver, not that the data write processing has completed. The completion of the data write can be checked by reading the return value (*USB_STS_WRITE_COMPLETE*) of the *R_USB_GetEvent* function.
3. Before calling this API, assign the PIPE number (*USB_PIPE1* to *USB_PIPE9*) to be used to the member *(pipe)* of the *usb_ctrl_t* structure. In USB Host mode, in order to identify the USB device to be accessed, assign the USB module number (*USB_IP0* or *USB_IP1*) to the member *(module)*, and assign the device address to the member *(address)*. If something other than *USB_IP0* or *USB_IP1* is assigned to the member *(module)*, then *USB_ERR_PARA* will be the return value.
4. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member *(module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
5. If something other than *USB_PIPE1* through *USB_PIPE9* is assigned to the member *(pipe)* of the *usb_ctrl_t* structure, then *USB_ERR_PARA* will be the return value.
6. Do not assign a pointer to the auto variable (stack) area to the second argument (*p_buf*).
7. The size of area assigned to the second argument (*p_buf*) must be at least as large as the size specified for the third argument (*size*).
8. If 0 (zero) is assigned to the argument (*p_ctrl* or *p_buf*), then *USB_ERR_PARA* will be the return value.
9. It is not possible to repeatedly call the *R_USB_PipeWrite* function with the same value assigned to the member *(pipe)* of the *usb_crtl_t* structure. If the *R_USB_PipeWrite* function is called repeatedly, then *USB_ERR_BUSY* will be the return value. To call the *R_USB_PipeWrite* function more than once with the same value assigned to the member *(pipe)*, first check the *USB_STS_WRITE_COMPLETE* return value from the *R_USB_GetEvent* function, and then call the *R_USB_PipeWrite* function.

10. In CDC/HID Class, to perform a Bulk/Interrupt transfer, use the *R_USB_Write* function rather than this API. In Host Mass Storage Class, to perform data access to the MSC device, use the FAT (File Allocation Table) API rather than this API.
11. Assign nothing to the member *(type)* of the *usb_ ctrl_t* structure. Even if the device class type or something is assigned to the member *(type)*, it is ignored.
12. To transfer the data for a Control transfer, use the *R_USB_Write* function rather than this API.
13. This API can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

```
void usb_application( void )
{
    usb_ctrl_t ctrl;
                :
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                    :
            case USB_STS_READ_COMPLETE:
                    :
                ctrl.moudle = USB_IP0;
                ctrl.pipe = USB_PIPE2;
                R_USB_PipeWrite(&ctrl, g_buf, size);
                    :
            break;
            case USB_STS_WRITE_COMPLETE:
                    :
            break;
                    :
        }
    }
}
```

4.14 R_USB_PipeStop

Stop data read/write via specified pipe

Format

usb_err_t	R_USB_PipeStop(usb_ctrl_t *p_ctrl)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area

Return Value

USB_SUCCESS	Successfully completed (stop request completed)
USB_ERR_PARA	Parameter error
USB_ERR_BUSY	Stop request already in process for USB device with same device address.
USB_ERR_NG	Other error

Description

This function is used to terminate a data read/write operation.

Note

1. Before calling this API, specify the selected pipe number (*USB_PIPE0* to *USB_PIPE9*) in the *usb_ctrl_t* member *(pipe)*. When using two USB modules in the USB host mode, also specify the number of the selected USB module (*USB_IP0*/*USB_IP1*) in the member (*module)*. If something other than *USB_IP0* or *USB_IP1* is assigned to the member (*module)*, then *USB_ERR_PARA* will be the return value. In USB Peripheral mode, no assignment to the members (*address* and *module*) is required. If assignment is performed, it is ignored.
2. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member (*module)*. If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
3. If something other than *USB_PIPE1* through *USB_PIPE9* is assigned to the member (*pipe)* of the *usb_ctrl_t* structure, then *USB_ERR_PARA* will be the return value.
4. In USB Host mode, *USB_ERR_NG* will be the return value when this API can not stop the data read/write request.
5. Assign nothing to the member (*type)* of the *usb_ ctrl_t* structure. Even if the device class type or something is assigned to the member (*type)*, it is ignored.
6. This API can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

```
void usb_application( void )
{
    usb_ctrl_t ctrl;
                        :
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                        :
            case USB_STS_DETACH:
                        :
                ctrl.module = USB_IP0;
                ctrl.pipe = USB_PIPE1;
                R_USB_PipeStop( &ctrl );
                        :
            break;
                        :
        }
    }
}
```

4.15 R_USB_GetUsePipe

Get used pipe number from bit map

Format

usb_err_t	R_USB_GetUsePipe(usb_ctrl_t *p_ctrl, uint16_t *p_pipe)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
p_pipe	Pointer to area that stores the selected pipe number (bit map information)

Return Value

USB_SUCCESS	Successfully completed
USB_ERR_PARA	Parameter error
USB_ERR_NG	Other error

Description

Get the selected pipe number (number of the pipe that has completed initalization) via bit map information. The bit map information is stored in the area specified in argument (*p_pipe)*. Based on the information (*module* member and *address* member) assigned to the *usb_ctrl_t* structure, obtains the PIPE information of that USB device.

The relationship between the pipe number specified in the bit map information and the bit position is shown below.

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
--	--	--	--	--	--	PIPE9	PIPE8	PIPE7	PIPE6	PIPE5	PIPE4	PIPE3	PIPE2	PIPE1	PIPE0
0	0	0	0	0	0	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	1

0:Not used, 1: Used

For example, when PIPE1, PIPE2, and PIPE8 are used, the value “0x0107” is set in the area specified in argument (*p_pipe).*

Note

1. In USB Host mode, before calling this API, assign the device address of the USB device whose Pipe information is to be obtained, and the USB module number (*USB_IP0/USB_IP1*) connected to that USB device, to the module (*address* and *module*) of the usb_ctrl_t structure. If something other than *USB_IP0* or *USB_IP1* is assigned to the member (*module*), then *USB_ERR_PARA* will be the return value.
2. If the MCU being used only supports one USB module, then do not assign USB_IP1 to the member (*module*). If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
3. In USB Peripheral mode, assign *USB_NULL* to the first argument (*p_ctrl*).
4. Bit map information b0(PIPE0) is always set to "1".
5. This API can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

1. **In the case of USB Host mode**

```
void usb_application( void )
{
    uint16_t usepipe;
    usb_ctrl_t ctrl;
                    :
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                    :
            case USB_STS_CONFIGURED:
                    :
                ctrl.module = USB_IP0;
                ctrl.address = adr;
                R_USB_GetUsePipe(&ctrl, &usepipe);
                    :
            break;
                    :
        }
    }
}
```

2. **In the case of USB Peripheral mode**

```
void usb_application( void )
{
    uint16_t usepipe;
    usb_ctrl_t ctrl;

    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
                    :
            case USB_STS_CONFIGURED:
                    :
                R_USB_GetUsePipe((usb_ctrl_t *)USB_NULL, &usepipe);
                    :
            break;
                    :
        }
    }
}
```

4.16 R_USB_GetPipeInfo

Get pipe information for specified pipe

Format

usb_err_t	R_USB_GetPipeInfo(usb_ctrl_t *p_ctrl, usb_pipe_t *p_info)

Arguments

p_ctrl	Pointer to usb_ctrl_t structure area
p_info	Pointer to usb_pipe_t structure area

Return Value

USB_SUCCESS	Successfully completed
USB_ERR_PARA	Parameter error
USB_ERR_NG	Other error

Description

This function gets the following pipe information regarding the pipe specified in the argument (*p_ctrl*) member (*pipe)*: endpoint number, transfer type, transfer direction and maximum packet size. The obtained pipe information is stored in the area specified in the argument (*p_info*).

Note

1. Before calling this API, specify the pipe number (*USB_PIPE1* to *USB_PIPE9*) in the *usb_ctrl_t* structure member *(pipe)*. When using two USB modules in the USB host mode, also specify the USB module number in the member *(module)*.
2. In USB Host mode, before calling this API, assign the device address of the USB device whose Pipe information is to be obtained, and the USB module number (*USB_IP0/USB_IP1*) connected to that USB device, to the members (*address* and *module*) of the *usb_ctrl_t* structure. If something other than *USB_IP0* or *USB_IP1* is assigned to the member (*module*), then *USB_ERR_PARA* will be the return value.
3. If the MCU being used only supports one USB module, then do not assign *USB_IP1* to the member (*module*). If *USB_IP1* is assigned, then *USB_ERR_PARA* will be the return value.
4. In USB Peripheral mode, no assignment to the members (*address* and *module*) is required.
5. Refer to chapter **8.5, usb_pipe_t structure** for details on the *usb_pipe_t* structure.
6. This function can be called when the USB device is in the configured state. When the API is called in any other state, *USB_ERR_NG* is returned.

Example

```
void usb_host_application( void )
{
        usb_pipe_t info;
        usb_ctrl_t ctrl;
                                :
        while (1)
        {
                switch (R_USB_GetEvent(&ctrl))
                {
                                :
                        case USB_STS_CONFIGURED:
                                :
                                ctrl.pipe = USB_PIPE3;
                                ctrl.module = USB_IP1;
                                ctrl.address= address;
                                R_USB_GetPipeInfo( &ctrl, &info );
                                :
                        break;
                                :
                }
        }
}
```

5. Return Value of R_USB_GetEvent Function

The return values for the *R_USB_GetEvent* function are listed below. Make sure you describe a program in the application program to be triggered by each return value from the *R_USB_GetEvent* function.

Return Value	Description	Host	Peri
USB_STS_DEFAULT	USB device has transitioned to default state.	×	○
USB_STS_CONFIGURED	USB device has transitioned to configured state.	○	○
USB_STS_SUSPEND	USB device has transitioned to suspend state.	×	○
USB_STS_RESUME	USB device has returned from suspend state.	○	○
USB_STS_DETACH	USB device has been detached from USB host.	○	○
USB_STS_REQUEST	USB device received USB request (Setup).	×	○
USB_STS_REQUEST_COMPLETE	USB request data transfer/receive is complete; device has transitioned to status stage.	○	○
USB_STS_READ_COMPLETE	USB data read processing is complete.	○	○
USB_STS_WRITE_COMPLETE	USB data write processing is complete.	○	○
USB_STS_BC	Attachment of USB device that supports battery charging function detected.	○	×
USB_STS_OVERCURRENT	Overcurrent detected.	○	×
USB_STS_NOT_SUPPORT	Unsupported USB device has been connected.	○	×
USB_STS_NONE	No USB-related events.	○	○

5.1 USB_STS_DEFAULT

When the *R_USB_GetEvent* function is called after the USB device has transitioned to the default state, the function sends *USB_STS_DEFAULT* as the return value.

5.2 USB_STS_CONFIGURED

When the *R_USB_GetEvent* function is called after the USB device has transitioned to the configured state, the function sends *USB_STS_CONFIGURED* as the return value.

module	:	The module number of the USB module that has transitioned to the Configured state (USB Host mode only).
type	:	Device class type (USB host mode only) when USB device has transitioned to configured state.
address	:	Device address (USB host mode only) when USB device has transitioned to configured state.

5.3 USB_STS_SUSPEND

When the *R_USB_GetEvent* function is called after the USB device has transitioned to the suspend state, the function sends *USB_STS_SUSPEND* as the return value.

5.4 USB_STS_RESUME

When the *R_USB_GetEvent* function is called after USB device in the suspend state resumes by the resume signal, the function sends *USB_STS_RESUME* as the return value.

Note:

In USB host mode, when the *R_USB_GetEvent* function is called after resuming by RemoteWakeUp signal from HID device, the function sends *USB_STS_RESUME* as the return value.

5.5 USB_STS_DETACH

When the *R_USB_GetEvent* function is called after the USB device has been detached from the USB host, the function sends *USB_STS_DETACH* as the return value. In USB host mode, information is also set in the following *usb_ctrl_t* structure member.

module	:	USB module number of detached USB module (in USB host mode only)
address	:	Device address of the detached USB device (in USB host mode only)

5.6 USB_STS_REQUEST

When the *R_USB_GetEvent* function is called after the USB device has received a USB request (Setup), the function sends *USB_STS_REQUEST* as the return value. Information is also set in the following *usb_ctrl_t* structure member.

setup	:	Received USB request information (8 bytes)

Note:

1. When a request has been received for support of the no-data control status stage, even if the *R_USB_GetEvent* function is called, *USB_STS_REQUEST_COMPLETE* is sent as the return value instead of *USB_STS_REQUEST*.
2. For more details on USB request information (8 bytes) stored in member (*setup)*, refer to chapter **8.2, usb_setup_t structure**.

5.7 USB_STS_REQUEST_COMPLETE

After the status stage of a control transfer is complete and transition to the idle stage has occurred, if the *R_USB_GetEvent* function is called, then *USB_STS_REQUEST_COMPLETE* will be the return value. In addition to this, the following member of the *usb_ctrl_t* structure also has information.

module	:	USB module number of completed the request (in USB host mode only)
address	:	Device address of USB device of completed the request (in USB host mode only)
status	:	Sets either USB_ACK / USB_STALL

Note:

When a request has been received for support of the no-data control status stage, USB request information (8 bytes) is stored in the *usb_ctrl_t* structure member (*setup*). For more details on USB request information (8 bytes) stored in member (*setup*), refer to chapter **8.2, usb_setup_t structure**.

5.8 USB_STS_READ_COMPLETE

When the *R_USB_GetEvent* function is called after a data read has been completed in the *R_USB_Read* function, *USB_STS_READ_COMPLETE* is sent as the return value. Information is also set in the following *usb_ctrl_t* structure member.

module	:	USB module number of completed data read (in USB host mode only)
address	:	Device address of USB device of completed data read (in USB host mode only)
type	:	Device class type of completed data read (only set when using R_USB_Read function)
size	:	Size of read data
pipe	:	Pipe number of completed data read
status	:	Read completion error information

Note:

1. In USB host mode, device address of USB device of completed data read is set in the member (*address)* and the USB module number (*USB_IP0 / USB_IP1*) of the connected USB device is set in the member (*module).*
2. In the case of the R_USB_PipeRead function, the member (*pipe*) has the PIPE number (*USB_PIPE1* to *USB_PIPE9*) for which data read is completed. In the case of the *R_USB_Read* function, *USB_NULL* is set to the member (*pipe*).
3. For details on device class type, refer to chapter **6, Device Class Types**.
4. The member (*status*) has the read completion error information. The error information set to this member is as follows.

USB_SUCCESS	:	Data read successfully completed
USB_ERR_OVER	:	Received data size over
USB_ERR_SHORT	:	Received data size short
USB_ERR_NG	:	Data reception failed

(1). Even if the reception request size is less than MaxPacketSize × n, if MaxPacketSize × n bytes of data are received, then *USB_ERR_OVER* is set.

For example, if MaxPacketSize is 64 bytes, the specified reception request size is 510 bytes (less than MaxPacketSize × n), and the actual received data size is 512 bytes (MaxPacketSize × n), then *USB_ERR_OVER* is set.

(2). If the reception request size is less than MaxPacketSize × n and the actual received data size is less than this reception request size, then *USB_ERR_SHORT* is set.

For example, if MaxPacketSize is 64 bytes, the specified reception request size is 510 bytes, and the actual received data size is 509 bytes, then *USB_ERR_SHORT* is set.

5.9 USB_STS_WRITE_COMPLETE

When the *R_USB_GetEvent* function is called after a data write has been completed in the *R_USB_Write* function, *USB_STS_WRITE_COMPLETE* is sent as the return value. Information is also set in the following *usb_ctrl_t* structure member.

module	:	USB module number of completed data write (in USB host mode only)
address	:	Device address of USB device of completed data write (in USB host mode only)
type	:	Device class type of completed data write (only set when using R_USB_Write function)
pipe	:	Pipe number of completed data write
status	:	Write completion error information

Note:

1. For *R_USB_Write* function: class type is set in the *usb_ctrl_t* structure member (*type*) and *USB_NULL* is set in the member (*pipe*).
2. In the case of *R_USB_PipeWrite* function, the member (*pipe*) has the PIPE number (*USB_PIPE1* to *USB_PIPE9*) for which data write has been completed. In the case of the *R_USB_Write* function, *USB_NULL* is set to the member (*pipe*).
3. For details on device class type, refer to chapter **6, Device Class Types**.
4. The member (*status*) has the write completion error information. The error information set to this member is as follows.

USB_SUCCESS	:	Data write successfully completed
USB_ERR_NG	:	Data transmission failed

5.10 USB_STS_BC

If the *R_USB_GetEvent* function is called after connecting to the USB device/USB Host that supports the Battery Charging function is detected, then *USB_STS_BC* will be the return value. Information is also set in the following *usb_ctrl_t* structure member.

module	:	USB module which USB device supports Battery Charging function is connected to (USB Host mode only)

5.11 USB_STS_OVERCURRENT

In USB Host mode, if the *R_USB_GetEvent* function is called after overcurrent is detected, then *USB_STS_OVERCURRENT* will be the return value. Information is also set in the following *usb_ctrl_t* structure member.

module	:	USB module number of detected overcurrent (USB_IP0 / USB_IP1

5.12 USB_STS_NOT_SUPPORT

In USB Host mode, if the *R_USB_GetEvent* function is called after an unsupported USB device is connected, then *USB_STS_NOT_SUPPORT* will be the return value.

5.13 USB_STS_NONE

When the *R_USB_GetEvent* function is called in the “no USB-related event” status, *USB_STS_NONE* is sent as the return value. Information is also set in the following *usb_ctrl_t* structure member.

status	:	USB device status

6. Device Class Types

The device class types assigned to the member(*type*) of the *usb_ctrl_t* and *usb_info_t* structures are as follows. Please specify the device class supported by your system.

Device class type	Description
USB_HCDC	Host Communication Device Class
USB_HCDCC	Host Communication Device Class (Control Class)
USB_HHID	Host Human Interface Device Class
USB_HMSC	Host Mass Storage Device Class
USB_PCDC	Peripheral Communication Device Class
USB_PCDCC	Peripheral Communication Device Class (Control Class)
USB_PHID	Peripheral Human Interface Device Class
USB_PMSC	Peripheral Mass Storage Device Class
USB_HVNDR	Host Vendor Class
USB_PVNDR	Peripheral Vendor Class

Note:

1. Host Communication Device Class: When transmitting data in a bulk transfer, specify *USB_HCDC* in the *usb_ctrl_t* structure member (*type)*. When transmitting data in an interrupt transfer, specify *USB_HCDC* in the *usb_ctrl_t* structure member (*type)*.
2. Peripheral Communication Device Class: When transmitting data in a bulk transfer, specify *USB_PCDC* in the *usb_ctrl_t* structure member (*type)*. When transmitting data in an interrupt transfer, specify *USB_PCDCC* in the *usb_ctrl_t* structure member (*type)*.
3. For an application program, do not assign *USB_HMSC*, *USB_PMSC*, *USB_HVND*, and *USB_PVND* to the member (*type)* of the *usb_ctrl_t* structure.

7. Configuration

7.1 Common Configurations

7.1.1 r_usb_basic_config.h

1. Common Settings in USB Host/USB Peripheral Mode

Perform settings for the definitions below in both USB Host and USB Peripheral modes.

(1). USB operating mode setting

Set the operating mode (Host/Peripheral) of the USB module for the definition of *USB_CFG_MODE*.

a. USB Host mode

Set *USB_CFG_HOST* for the definition of *USB_CFG_MODE*.

```
#define         USB_CFG_MODE            USB_CFG_HOST
```

b. USB Peripheral mode

Set *USB_CFG_PERI* for the definition of *USB_CFG_MODE*.

```
#define         USB_CFG_MODE            USB_CFG_PERI
```

(2). Argument check setting

Specify whether to perform argument checking for all of the APIs listed in chapter **4, API Functions**.

```
#define         USB_CFG_PARAM_CHECKING   USB_CFG_ENABLE      // Checks arguments.
#define         USB_CFG_PARAM_CHECKING   USB_CFG_DISABLE     // Does not check
                                                             arguments.
```

(3). Device class setting

Enable the definition of the USB driver to be used among the definitions below.

```
#define       USB_CFG_HCDC_USE          // Host Communication Device Class
#define       USB_CFG_HHID_USE          // Host Human Interface Device Class
#define       USB_CFG_HMSC_USE          // Host Mass Storage Class
#define       USB_CFG_HVNDR_USE         // Host Vendor Class
#define       USB_CFG_PCDC_USE          // Peripheral Communication Device Class
#define       USB_CFG_PHID_USE          // Peripheral Human Interface Device Class
#define       USB_CFG_PMSC_USE          // Peripheral Mass Storage Class
#define       USB_CFG_PVNDR_USE         // Peripheral Vendor Class
```

(4). DTC use setting

Specify whether to use the DTC.

```
#define        USB_CFG_DTC          USB_CFG_ENABLE    // Uses DTC
#define        USB_CFG_DTC          USB_CFG_DISABLE   // Does not use DTC
```

Note:

If *USB_CFG_ENABLE* is set for the definition of *USB_CFG_DTC*, be sure to set *USB_CFG_DISABLE* for the definition of *USB_CFG_DMA* in (5) below.

(5). DMA use setting

Specify whether to use the DMA.

```
#define        USB_CFG_DMA          USB_CFG_ENABLE    // Uses DMA.
#define        USB_CFG_DMA          USB_CFG_DISABLE   // Does not use DMA.
```

Note:

a. If *USB_CFG_ENABLE* is set for the definition of *USB_CFG_DMA*, be sure to set *USB_CFG_DISABLE* for the definition of *USB_CFG_DTC* in (4) above.

b. If *USB_CFG_ENABLE* is set for the definition of *USB_CFG_DMA*, set the DMA Channel number for the definition in (6) below.

(6). DMA Channel setting

If *USB_CFG_ENABLE* is set in (5) above, set the DMA Channel number to be used.

```
#define    USB_CFG_USB0_DMA_TX    DMA Channel number    // Transmission setting for
                                                        USB0 module
#define    USB_CFG_USB0_DMA_RX    DMA Channel number    // Transmission setting for
                                                        USB0 module
#define    USB_CFG_USB1_DMA_TX    DMA Channel number    // Transmission setting for
                                                        USB1 module
#define    USB_CFG_USB1_DMA_RX    DMA Channel number    // Transmission setting for
                                                        USB1 module
```

Note:

a. Set one of the DMA Channel numbers from *USB_CFG_CH0* to *USB_CFG_CH3* or *USB_CFG_CH7*. Do not set the same DMA Channel number.

b. If DMA transfer is not used, set *USB_CFG_NOUSE* as the DMA Channel number.

c. Set the DMA Channel numbers (see below) for the definitions for sending and receiving. It is impossible to set a DMA Channel number for only one of the definitions for sending and receiving.

   Example) When the DMA transfer is set for the USB0 module

```
#define    USB_CFG_USB0_DMA_TX    USB_CFG_CH0
#define    USB_CFG_USB0_DMA_RX    USB_CFG_CH4
```

d. To perform the DMA transfer using two USB modules (USB0 and USB1), use four DMA Channels.

(7). Setting Battery Charging (BC) function

Set the Battery Charging function to be enabled or disabled as the following definition. Set *USB_CFG_ENABLE* as the definition below in order to use the Battery Charging function.

```
#define    USB_CFG_BC    USB_CFG_ENABLE     // Uses BC function.
#define    USB_CFG_BC    USB_CFG_DISABLE    // Does not use BC function.
```

Note:

In the case of a USB module other than USBAa/USBA module, this definition is ignored.

(8). Endian setting

Specify the Endian type for the definition below.

```
#define    USB_CFG_ENDIAN    USB_CFG_LITTLE    // Little Endian
#define    USB_CFG_ENDIAN    USB_CFG_BIG       // Big Endian
```

(9). PLL clock frequency setting

Set the PLL clock source frequency for the definition below.

```
#define    USB_CFG_CLKSEL    USB_CFG_24MHZ    // Set to 24 MHz
#define    USB_CFG_CLKSEL    USB_CFG_20MHZ    // Set to 20 MHz
#define    USB_CFG_CLKSEL    USB_CFG_OTHER    // Set to other than 24/20 MHz
```

Note:

a. In the case of a USB module other than USBAa/USBA module, this definition is ignored.

b. The USBAa or USBA module is a USB module used in the RX71M or RX64M.

c. To input a clock other than a 24-MHz or 20-MHz clock to the XTAL pin, set *USB_CFG_OTHE*R for the definition of *USB_CFG_CLKSEL*. If *USB_CFG_OTHER* is set, the USBAa/USBA module operates in Classic (CL) only mode. For information on CL only mode, refer to the RX71M/RX64M hardware manual.

(10). CPU bus wait setting

Assign the value to be set for the BUSWAIT register in the USBAa/USBA module as the definition of *USB_CFG_BUSWAIT*.

```
#define    USB_CFG_BUSWAIT    7    // Set to 7 wait cycles
```

Note:

a. For the calculation of the value to be set for *USB_CFG_BUSWAIT*, refer to the chapter of the BUSWAIT register in the RX71M/RX64M hardware manual.

b. With regard to the USB module other than the USBAa/USBA module, this definition is ignored.

c. The USBAa or USBA module is a USB module used in the RX71M or RX64M.

2. Settings in USB Host Mode

To make a USB module to work as a USB Host, set the definitions below according to the system to be used.

(1). Setting power source IC for USB Host

Set the VBUS output of the power source IC for the USB Host being used to either Low Assert or High Assert. For Low Assert, set *USB_CFG_LOW* as the definition below, and for High Assert, set *USB_CFG_HIGH* as the definition below.

```
#define    USB_CFG_VBUS    USB_CFG_HIGH    // High Assert
#define    USB_CFG_VBUS    USB_CFG_LOW     // Low Assert
```

(2). Setting USB port operation when using Battery Charging (BC) function

Set the Dedicated Charging Port (DCP) to be enabled or disabled as the following definition. If the BC function is being implemented as the Dedicated Charging Port (DCP), then set *USB_CFG_ENABLE* as the definition below. If *USB_CFG_DISABLE* is set, the BC function is implemented as the Charging Downstream Port (CDP).

```
#define    USB_CFG_DCP    USB_CFG_ENABLE     // DCP enabled.
#define    USB_CFG_DCP    USB_CFG_DISABLE    // DCP disabled.
```

Note:

If *USB_CFG_ENABLE* is set for this definition, then set *USB_CFG_ENABLE* for the definition of *USB_CFG_BC* in above.

(3). Setting Compliance Test mode

Set Compliance Test support for the USB Embedded Host to be enabled or disabled as the following definition. To perform the Compliance Test, set *USB_CFG_ENABLE* as the definition below. When not performing the Compliance Test, set *USB_CFG_DISABLE* as the definition below.

```
#define    USB_CFG_COMPLIANCE    USB_CFG_ENABLE     // Compliance Test supported.
#define    USB_CFG_COMPLIANCE    USB_CFG_DISABLE    // Compliance Test not supported.
```

(4). Setting a Targeted Peripheral List (TPL)

Set the number of the USB devices and the VID and PID pairs for the USB device to be connected as necessary as the following definition. For a method to set the TPL, see chapter **3.6, How to Set the Target Peripheral List (TPL)**.

```
#define    USB_CFG_TPLCNT    Number of the USB devices to be connected.
#define    USB_CFG_TPL       Set the VID and PID pairs for the USB device to be
                             connected.
```

(5). Setting a Targeted Peripheral List (TPL) for USB Hub

Set the number of the USB Hubs and the VID and PID pairs for the USB Hubs to be connected as the following definition. For a method to set the TPL, see chapter **3.6, How to Set the Target Peripheral List (TPL)**.

```
#define    USB_CFG_HUB_TPLCNT    Set the number of the USB Hubs to be connected.
#define    USB_CFG_HUB_TPL       Set the VID and PID pairs for the USB Hub to be
                                 connected.
```

(6). Setting Hi-speed Embedded Host Electrical Test

Set Hi-speed Embedded Host Electrical Test support to be enabled or disabled as the following definition. To perform the Hi-speed Embedded Host Electrical Test, set *USB_CFG_ENABLE* as the definition below.

```
#define    USB_CFG_ELECTRICAL    USB_CFG_ENABLE    // HS Electrical Test supported
```

```
#define    USB_CFG_ELECTRICAL    USB_CFG_DISABLE    // HS Electrical Test not supported
```

Note:

a. If *USB_CFG_ENABLE* is set for this definition, then set *USB_CFG_ENABLE* for the definition of *USB_CFG_COMPLIANCE* in (3) above.

b. In the case of a USB module other than USBAa module, this definition is ignored.

3. Settings in USB Peripheral Mode

To make a USB module to work as a USB Peripheral, set the definitions below according to the system to be used.

(1). USB module selection setting

Set the USB module number to be used for the definition of *USB_CFG_USE_USBIP*.

```
#define    USB_CFG_USE_USBIP    USB_CFG_IP0    // Uses USB0 module
#define    USB_CFG_USE_USBIP    USB_CFG_IP1    // Uses USB1 module
```

Note:

If the MCU being used only supports one USB module, then set *USB_CFG_IP0* for the definition of *USB_CFG_USE_USBIP*.

(2). Setting class request

Set whether the received class request is supported. If *USB_CFG_ENABLE* (supported) is set, then the USB driver will notify the reception of the class request to the application program. If *USB_CFG_DISABLE* (not supported) is set, then the USB driver will respond a STALL to the class request.

```
#define    USB_CFG_CLASS_REQUEST    USB_CFG_ENABLE     // Supported
#define    USB_CFG_CLASS_REQUEST    USB_CFG_DISABLE    // Not supported
```

Note:

a. Check the return value (*USB_STS_REQUEST*) of *R_USB_GetEvent* function when confirming whether USB driver receive the class request or not.

b. Even if *USB_CFG_DISABLE* is set, USB driver return the value "1" to GetMaxLun class request of Mass storage class.

(3). Setting power saving function

Set the power saving function to be enabled or disabled as the definition below. If *USB_CFG_ENABLE* is set as the definition below, then when there is a transition to suspend state or detach state, the USB driver will transition the MCU to power saving mode.

```
#define    USB_CFG_LPW    USB_CFG_ENABLE     // Power saving function enabled.
#define    USB_CFG_LPW    USB_CFG_DISABLE    // Power saving function disabled.
```

4. Other Definitions

In addition to the above, the following definitions (1) through (2) are also provided in *r_usb_basic_config.h*. Recommended values have been set for these definitions, so only change them when necessary.

(1). DBLB bit setting

Set or clear the DBLB bit in the pipe configuration register (PIPECFG) of the USB module using the following definition.

```
#define    USB_CFG_DBLB    USB_CFG_DBLBON     // DBLB bit set.
#define    USB_CFG_DBLB    USB_CFG_DBLBOFF    // DBLB bit cleared.
```

(2). CNTMD bit setting (USBA/USBAa module only)

Set or clear the CNTMD bit in the pipe configuration register (PIPECFG) of the USB module using the following definition.

```
#define    USB_CFG_CNTMD    USB_CFG_CNTMDON     // CNTMD bit set.
#define    USB_CFG_CNTMD    USB_CFG_CNTMDOFF    // CNTMD bit cleared.
```

Note:

1. The setting of the DBLB and CNTMD bits above is performed for all the pipes being used. Therefore, in this configuration, it is not possible to perform the pipe-specific settings for these bits.
2. For details on the pipe configuration register (PIPECFG), refer to the MCU hardware manual.
3. Be sure to set SHTNAK bit.

7.2 Configuration for Device Classes

7.2.1 r_usb_hcdc_config.h (for Host Communication Device Class Driver)

1. Setting connection of multiple CDC devices

To simultaneously connect multiple CDC devices and perform USB communication, set *USB_CFG_ENABLE* as the definition below. If multiple CDC devices are not connected simultaneously, then set *USB_CFG_DISABLE*.

```
#define    USB_CFG_HCDC_MULTI    USB_CFG_ENABLE     // Multiple connection supported
#define    USB_CFG_HCDC_MULTI    USB_CFG_DISABLE    // Multiple connection not
                                                    supported
```

2. Setting CDC class

Specify the device class ID of the CDC device to be connected.

```
#define    USB_CFG_HCDC_IFCLS    USB_CFG_CDC        // CDC class supported device
#define    USB_CFG_HCDC_IFCLS    USB_CFG_VEN        // Vendor class device
```

Note:

With regard to the USB serial conversion device in the marketplace, the device class ID may be the Vendor class. Check the CDC device specifications before use. If the device class is Vendor class, then set *USB_CFG_VEN*.

3. Setting pipe to be used

Set the pipe number to use for data transfer.

(1). Bulk IN/OUT transfer

Set the pipe number (PIPE1 to PIPE5) to use for Bulk IN/OUT transfer. Do not set the same pipe number.

```
#define    USB_CFG_HCDC_BULK_IN         Pipe number (USB_PIPE1 to USB_PIPE5)
#define    USB_CFG_HCDC_BULK_OUT        Pipe number (USB_PIPE1 to USB_PIPE5)
#define    USB_CFG_HCDC_BULK_IN2        Pipe number (USB_PIPE1 to USB_PIPE5)
#define    USB_CFG_HCDC_BULK_OUT2       Pipe number (USB_PIPE1 to USB_PIPE5)
```

(2). Interrupt IN transfer

Set the pipe number (PIPE6 to PIPE9) to use for Interrupt IN transfer. Do not set the same pipe number. If the USB Hub is being used, then PIPE9 cannot be set as the following definitions.

```
#define    USB_CFG_HCDC_INT_IN          Pipe number (USB_PIPE6 to USB_PIPE9)
#define    USB_CFG_HCDC_INT_IN2         Pipe number (USB_PIPE6 to USB_PIPE9)
```

Note:

a. Only if *USB_CFG_ENABLE* is set for the definition of *USB_CFG_HCDC_MULTI* in 1 above, set the pipe number for the definitions of *USB_CFG_HCDC_BULK_IN2, USB_CFG_HCDC_BULK_OUT2,* and *USB_CFG_HCDC_INT_IN2*.

b. If USB_CFG_DISABLE is set for the definition of *USB_CFG_HCDC_MULTI*, set *USB_NULL* for the definitions of *USB_CFG_HCDC_BULK_IN2, USB_CFG_HCDC_BULK_OUT2,* and *USB_CFG_HCDC_INT_IN2.*

7.2.2 r_usb_hhid_config.h (for Host Human Interface Device Class Driver)

1. Setting pipe to be used

Set the pipe number (PIPE6 to PIPE9) to use for Interrupt IN transfer. Do not set the same pipe number. If the USB Hub is being used, then PIPE9 cannot be set as the following definitions.

```
#define    USB_CFG_HHID_INT_IN          Pipe number (USB_PIPE6 to USB_PIPE9)
```

```
#define    USB_CFG_HHID_INT_IN2         Pipe number (USB_PIPE6 to USB_PIPE9)
#define    USB_CFG_HHID_INT_IN3         Pipe number (USB_PIPE6 to USB_PIPE9)
```

Note:

If no pipe number is required to be set for the definitions of *USB_CFG_HHID_INT_IN2* and *USB_CFG_HHID_INT_IN3*, then set *USB_NULL* as these definitions.

7.2.3 r_usb_pcdc_config.h (for Peripheral Communication Device Class Driver)

1. Setting pipe to be used

Set the pipe number to use for data transfer.

(1). Bulk IN/OUT transfer

Set the pipe number (PIPE1 to PIPE5) to use for Bulk IN/OUT transfer. Do not set the same pipe number for the definitions of *USB_CFG_PCDC_BULK_IN* and *USB_CFG_PCDC_BULK_OUT*.

```
#define    USB_CFG_PCDC_BULK_IN         Pipe number (USB_PIPE1 to USB_PIPE5)
#define    USB_CFG_PCDC_BULK_OUT        Pipe number (USB_PIPE1 to USB_PIPE5)
```

(2). Interrupt IN transfer

Set the pipe number (PIPE6 to PIPE9) to use for Interrupt IN transfer.

```
#define    USB_CFG_PCDC_INT_IN          Pipe number (USB_PIPE6 to USB_PIPE9)
```

7.2.4 r_usb_phid_config.h (for Peripheral Human Interface Device Class Driver)

1. Setting pipe to be used

Set the pipe number (PIPE6 to PIPE9) to use for Interrupt IN/OUT transfer. Do not set the same pipe number for the definitions of *USB_CFG_PHID_INT_IN* and *USB_CFG_PHID_INT_OUT*.

```
#define    USB_CFG_PHID_INT_IN          Pipe number (USB_PIPE6 to USB_PIPE9)
#define    USB_CFG_PHID_INT_OUT         Pipe number (USB_PIPE6 to USB_PIPE9)
```

Note:

For a system that does not support the OUT transfer, set *USB_NULL* as the definition of *USB_CFG_PHID_INT_OUT*.

7.2.5 r_usb_pmsc_config.h (for Peripheral Mass Storage Class Driver)

1. Setting pipe to be used

Set the pipe number (PIPE1 to PIPE5) to use for Bulk IN/OUT transfer. Do not set the same pipe number for the definitions of *USB_CFG_PMSC_BULK_IN* and *USB_CFG_PMSC_BULK_OUT*.

```
#define    USB_CFG_PMSC_BULK_IN         Pipe number (USB_PIPE1 to USB_PIPE5)
#define    USB_CFG_PMSC_BULK_OUT        Pipe number (USB_PIPE1 to USB_PIPE5)
```

8. Structures

This chapter describes the structures used in the application program.

8.1 usb_ctrl_t structure

The *usb_ctrl_t* structure is used for USB data transmission and other operations. The *usb_ctrl_t* structure can be used in all APIs listed in Table 4-1, excluding *R_USB_GetVersion*.

```
typdef union usb_ctrl {
        uint8_t         module;         /* Note 1 */
        uint8_t         address;        /* Note 2 */
        uint8_t         pipe;           /* Note 3 */
        uint8_t         type;           /* Note 4 */
        uint16_t        status;         /* Note 5 */
        uint32_t        size;           /* Note 6 */
        usb_set_up      setup;          /* Note 7 */
} usb_ctrl_t;
```

Note:

1. Member (*module)* is used to specify the USB module number.
2. Member (*address)* is used to specify the USB device address.
3. Member (*pipe)* is used to specify the USB module pipe number. For example, specify the pipe number when using the *R_USB_PipeRead* function or *R_USB_PipeWrite* function.
4. Member (*type)* is used to specify the device class type.
5. The USB device state or the result of a USB request command is stored in the member (*status*). The USB driver sets in this member. Therefore, except when initializing the *usb_crtl_t* structure area or processing an ACK/STALL response to a vendor class request, the application program should not write into this member. For processing an ACK/STALL response to a vendor class request, see **9.2.5, Processing ACK/STALL Response to Class Request**.
6. Member (*size*) is used to set the size of data that is read. The USB driver sets this member. Therefore, the application program should not write into this member.
7. Member (*setup*) is used to set the information about a class request.

8.2 usb_setup_t structure

The *usb_setup_t* structure is used when sending or receiving a USB class request. To send a class request to a USB device (in USB Host mode), assign to the members of the *usb_setup_t* structure the information for the class request to be sent. To obtain class request information from the USB Host (in USB Peripheral mode), refer to the members of the *usb_setup_t* structure.

```
typedef struct usb_setup {
        uint16_t        type                    /* Note 1 */
        uint16_t        value;                  /* Note 2 */
        uint16_t        index;                  /* Note 3 */
        uint16_t        length;                 /* Note 4 */
} usb_setup_t;
```

Note:

1. In USB Host mode, the value assigned to the member (*type*) is set to the USBREQ register, and in USB Peripheral mode, the value of the USBREQ register is set to the member (*type*).
2. In USB Host mode, the value assigned to the member (*value*) is set to the USBVAL register, and in USB Peripheral mode, the value of the USBVAL register is set to the member (*value*).
3. In USB Host mode, the value assigned to the member (*index*) is set to the USBINDX register, and in USB Peripheral mode, the value of the USBINDX register is set to the member (*index*).
4. In USB Host mode, the value assigned to the member (*length*) is set to the USBLENG register, and in USB Peripheral mode, the value of the USBLENG register is set to the member (*length*).
5. For information on the USBREQ, USBVAL, USBINDX, and USBLENG registers, refer to the MCU user's manual.

8.3 usb_cfg_t structure

The *usb_cfg_t* structure is used to register essential information such as settings to indicate use of USB host or USB peripheral as the USB module and to specify USB speed. This structure can only be used for the *R_USB_Open* function listed in **Table 4-1**.

```
typdef struct usb_cfg {
        uint8_t                 usb_mode;       /* Note 1 */
        uint8_t                 usb_speed;      /* Note 2 */
        usb_descriptor_t        *p_usb_reg;     /* Note 3 */
} usb_cfg_t;
```

Note:

1. Specify whether to use USB host or USB peripheral mode as the USB module in member (*usb_mode)*. To select USB host, set *USB_HOST*; to select USB peripheral, set *USB_PERI* in the member.
2. Specify the USB speed for USB module operations. Set "*USB_HS*" to select Hi-speed, "*USB_FS*" to select Full-speed.
3. Specify the *usb_descriptor_t* type pointer for the USB device in member (*p_usb_reg)*. Refer to chapter **8.4, usb_descriptor_t structure** for details on the *usb_descriptor_t* type. This member can only be set in USB peripheral mode. Even if it is set in USB host mode, the settings will be ignored.

8.4 usb_descriptor_t structure

The *usb_descriptor_t* structure stores descriptor information such as device descriptor and configuration descriptor. The descriptor information set in this structure is sent to the USB host as response data to a standard request during enumeration of the USB host. This structure is specified in the *R_USB_Open* function argument.

```
typdef struct usb_descriptor {
        uint8_t         *p_device;              /* Note 1 */
        uint8_t         *p_config_f;            /* Note 2 */
        uint8_t         *p_config_h;            /* Note 3 */
        uint8_t         *p_qualifier;           /* Note 4 */
        uint8_t         **p_string;             /* Note 5 */
} usb_descriptor_t;
```

Note:

1. Specify the top address of the area that stores the device descriptor in the member (*p_device)*.
2. Specify the top address of the area that stores the Full-speed configuration descriptor in the member (*p_config_f)*. Even when using Hi-speed, make sure you specify the top address of the area that stores the Full-speed configuration descriptor in this member.
3. Specify the top address of the area that stores the Hi-speed configuration descriptor in the member (*p_config_h)*. For Full-speed, specify *USB_NULL* to this member.
4. Specify the top address of the area that stores the qualifier descriptor in the member (*p_qualifier)*. For Full-speed, specify *USB_NULL* to this member.

5. Specify the top address of the string descriptor table in the member (*p_string)*. In the string descriptor table, specify the top address of the areas that store each string descriptor.

Ex. 1) Full-speed

```
usb_descriptor_t usb_descriptor =
{
    smp_device,
    smp_config_f,
    USB_NULL,
    USB_NULL,
    smp_string,
};
```

Ex. 2) Hi-speed

```
usb_descriptor_t usb_descriptor =
{
    smp_device,
    smp_config_f,
    smp_config_h,,
    smp_qualifier,
    smp_string,
};
```

8.5 usb_pipe_t structure

The USB driver sets information about the USB pipe (PIPE1 to PIPE9) in the *usb_pipe_t* structure. Use the *R_USB_GetPipeInfo* function to reference the pipe information set in the structure.

```
typdef struct usb_pipe {
    uint8_t     ep;         /* Note 1 */
    uint8_t     type;       /* Note 2 */
    uint16_t    mxps;       /* Note 3 */
} usb_pipe_t;
```

Note:

1. The endpoint number is set in member (*ep)*. The direction (IN/OUT) is set in the highest bit. When the highest bit is "1", the direction is IN, when "0", the direction is OUT.
2. The transfer type (bulk/interrupt) is set in member (*type)*. For a Bulk transfer, "*USB_BULK*" is set, and for an Interrupt transfer, "*USB_INT*" is set.
3. The maximum packet size is set in member (*mxps)*.

8.6 usb_info_t structure

The following information on the USB device is set for the *usb_info_t* structure by calling the *R_USB_GetInformation* function.

```
typedef struct usb_info {
    uint8_t     type;       /* Note 1 */
    uint8_t     speed;      /* Note 2 */
    uint8_t     status;     /* Note 3 */
    uint8_t     port;       /* Note 4 */
} usb_info_t;
```

Note:

1. In USB Host mode, the device class type of the connected USB device is set for the member (*type*). If the USB device is not connected, then *USB_NOT_CONNECT* is set. In USB Peripheral mode, the supporting device class type is set for the member (*type*). For information on the device class types, see **6, Device Class Types**. (In the case of PCDC, *USB_PCDC* is set in this member(*type*))
2. The USB speed (*USB_HS/USB_FS/USB_LS*) is set for the member (*speed*). In USB Host mode, if no USB device is connected, then *USB_NOT_CONNECT* is set.
3. One of the following states of the USB device is set for the member (*status*).

USB_STS_DEFAULT	:	Default state
USB_STS_ADDRESS	:	Address state (USB Peripheral only)
USB_STS_CONFIGURED	:	Configured state
USB_STS_SUSPEND	:	Suspend state
USB_STS_DETACH	:	Detach state

4. The following information of the Battery Charging (BC) function of the device conected to the port is set to the member (*port*).

USB_SDP	:	Standard Downstream Port
USB_CDP	:	Charging Downstream Port
USB_DCP	:	Dedicated Charging Port (USB Peripheral only)

8.7 usb_compliance_t structure

This structure is used when running the USB compliance test. The structure specifies the following USB-related information:

```
typedef struct usb_compliance {
    usb_ct_status_t     status;     /* Note 1 */
    uint16_t            vid;        /* Note 2 */
    uint16_t            pid;        /* Note 3 */
} usb_compliance_t;
```

Note:

1. The member status can be set to the following values to indicate the status of the connected USB device:

USB_CT_ATTACH	:	USB device attach detected
USB_CT_DETACH	:	USB device detach detected
USB_CT_TPL	:	Attach detected of USB device listed in TPL
USB_CT_NOTTPL	:	Attach detected of USB device not listed in TPL
USB_CT_HUB	:	USB hub connection detected
USB_CT_OVRCUR	:	Overcurrent detected
USB_CT_NORES	:	No response to control read transfer
USB_CT_SETUP_ERR	:	Setup transaction error occurred

2. The member vid is set to a value indicating the vendor ID of the connected USB device.
3. The member pid is set to a value indicating the product ID of the connected USB device.

9. USB Class Requests

This chapter describes how to process USB class requests. As standard requests are processed by the USB driver, they do not need to be included in the application program.

9.1 USB Host operations

9.1.1 USB request (setup) transfer

A USB request is sent to the USB device using the *R_USB_Write* function. The following describes the transfer procedure.

1. Set *USB_REQUEST* in the *usb_ctrl_t* structure member (*type).*
2. Set the USB request (setup: 8 bytes) in the *usb_ctrl_t* structure member (*setup)* area. Refer to chapter **8.2, usb_setup_t structure** for details on how to set member *(setup).*
3. If the request supports the control write data stage, store the transfer data in a buffer. If the request supports the control read data stage, reserve a buffer to store the data received from the USB device. Note: do not reserve the auto-variable (stack) area of the buffer.
4. Specify the data buffer top address in the second argument of the *R_USB_Write* function, and the data size in the third argument. If the request supports no-data control status stage, specify *USB_NULL* for both the second and third arguments.
5. Call the *R_USB_Write* function.

9.1.2 USB request completion

Confirm the completion of a USB request with the return value (*USB_STS_REQUEST_COMPLETE*) of the *R_USB_GetEvent* function. For a request that supports the control read data stage, the received data is stored in the area specified in the second argument of the *R_USB_Write* function.

Confirm the USB request results from the *usb_ctrl_t* structure member (*status)*, which is set as follows.

status	Description
USB_ACK	Successfully completed
USB_STALL	Stalled

9.1.3 USB request processing example

```
void usr_application (void )
{
    usb_ctrl_t ctrl;
    switch( R_USB_GetEvent( &ctrl ) )
    {
            /* Request setting processing to ctrl.setup */
                    :
            /* For request that supports control write data stage, set transfer data in g_buf area. */
                    :
            ctrl.type = USB_REQUEST;
            R_USB_Write(&ctrl, g_buf, size); /* Send USB request (Setup stage). */
            break;
        case USB_STS_REQUEST_COMPLETE: /* USB request completed. */
            if(USB_ACK == ctrl.status) /* Confirm results of USB request. */
            {
                /* For request that supports control read data stage, store receive data in g_buf area. */
                                :
            }
            break;
    }
}
```

9.2 USB Peripheral operations

9.2.1 USB request (Setup)

Confirm receipt of the USB request (Setup) sent by the USB host with the return value (*USB_STS_REQUEST*) of the *R_USB_GetEvent* function. The contents of the USB request (Setup: 8 bytes) are stored in the *usb_ctrl_t* structure member (*setup)* area. Refer to chapter **8.2, usb_setup_t structure** for a description of the settings for member (*setup).*

Note:

The return value of the *R_USB_GetEvent* function when a request that supports the no-data control status stage is received is *USB_STS_REQUEST_COMPLETE*, not *USB_STS_REQUEST*.

9.2.2 USB request data

The *R_USB_Read* function is used to receive data in the data stage and the *R_USB_Write* function is used to send data to the USB host. The following describes the receive and send procedures.

1. Receive procedure

(1). Set the *USB_REQUEST* in the *usb_ctrl_t* structure member (*type).*

(2). In the *R_USB_Read* function, specify the pointer to area that stores data in the second argument, and the requested data size in the third argument.

(3). Call the *R_USB_Read* function.

Note:

Confirm receipt of the request data with the return value (*USB_STS_REQUEST_COMPLETE*) of the *R_USB_GetEvent* function.

2. Send procedure

(1). Set *USB_REQUEST* in the *usb_ctrl_t* structure member (*type).*

(2). Store the data from the data stage in a buffer. In the *R_USB_Write* function, specify the top address of the buffer in the second argument, and the transfer data size in the third argument.

(3). Call the *R_USB_Write* function.

Note:

Confirm receipt of the request data with the return value (*USB_STS_WRITE_COMPLETE*) of the *R_USB_GetEvent* function. You can also confirm whether the *usb_ctrl_t* structure member (*type)* has been set to *USB_REQUEST*.

9.2.3 USB request results

For each class, if *USB_CFG_ENABLE* is set as the definition of the class request setting (example: *USB_CFG_PCDC_REQUEST*) in the configuration file (example: *r_usb_pcdc_config.h*), then this USB driver will always respond with an ACK to a received class request.

Note:

For a vendor class request, the USB driver does not respond with an ACK or STALL. An application program must respond with an ACK or STALL to the vendor class request. For how to respond with an ACK or STALL, see **9.2.5, Processing ACK/STALL Response to Class Request.**

9.2.4 Example USB request processing description

1. Request that supports control read data stage

```
void usr_application (void )
{
    usb_ctrl_t ctrl;
    switch( R_USB_GetEvent( &ctrl ) )
    {
                    :
        case USB_REQUEST: /* Receive USB request */
            /* ctrl.setup analysis processing*/
                    :
            /* data setup processing */
                    :
            ctrl.type = USB_REQUEST;
            R_USB_Write(&ctrl, g_buf, size); /* data (data stage) send request */
            break;
        case USB_STS_REQUEST_COMPLETE:
            if(USB_ACK == ctrl.status) /* Confirm USB request results */
            {
                    :
            }
            break;
    }
}
```

2. Request that supports control write data stage

```
void usr_application (void )
{
    usb_ctrl_t ctrl;
    switch( R_USB_GetEvent( &ctrl ) )
    {
                    :
        case USB_REQUEST: /* Receive USB request */
            /* ctrl.setup analysis processing */
                    :
            ctrl.type = USB_REQUEST;
            R_USB_Read(&ctrl, g_buf, size); /* data (data stage) receive request */
            break;
        case USB_STS_REQUEST_COMPLETE:
                    :
            break;
    }
}
```

3. Request that supports no-data control status stage

```
void usr_application (void )
{
    usb_ctrl_t ctrl;
    switch( R_USB_GetEvent( &ctrl ) )
    {
                    :
        case USB_STS_REQUEST_COMPLETE:
            /* ctrl.setup analysis processing */
                    :
            break;
    }
}
```

9.2.5 Processing ACK/STALL Response to Class Request

When it is necessary to respond with ACK or STALL to a class request, assign *USB_REQUEST* to the member(*type*) of the *usb_ctrl_t* structure, and either *USB_ACK* or *USB_STALL* to the member (*status*), and call the *R_USB_Write* function. Assign *USB_NULL* to both the second and third arguments of the *R_USB_Write* function. The completion of transmission of ACK/STALL can be checked by reading the *USB_STS_REQUEST_COMPLETE* return value of the *R_USB_GetEvent* function. At this time, check also that *USB_REQUEST* has been set for the member (*type*) of the *usb_ctrl_t* structure.

1. Example of processing STALL response

```
void usr_application (void )
{
    usb_ctrl_t ctrl;
    switch( R_USB_GetEvent( &ctrl ) )
    {
                    :
        case USB_STS_REQUEST:
            /* ctrl.setup analysis processing */
                    :
            ctrl.type = USB_REQUEST:
            ctrl.status = USB_STALL;
            R_USB_Write(&ctrl, (uint8_t *)USB_NULL, (uint32_t)USB_NULL);
            break;
        case USB_STS_REQUEST_COMPLETE:
            if( USB_REQUEST == ctrl.type )
            {
                        :
            }
            break;
    }
}
```

2. Example of processing ACK response

```
void usr_application (void )
{
    usb_ctrl_t ctrl;
    switch( R_USB_GetEvent( &ctrl ) )
    {
                    :
        case USB_STS_REQUEST:
            /* ctrl.setup analysis processing */
                    :
            ctrl.type = USB_REQUEST:
            ctrl.status = USB_ACK;
            R_USB_Write(&ctrl, (uint8_t *)USB_NULL, (uint32_t)USB_NULL);
            break;
        case USB_STS_REQUEST_COMPLETE:
            if( USB_REQUEST == ctrl.type )
            {
                        :
            }
            break;
    }
}
```

10. DTC/DMA Transfer

10.1 Basic Specification

The specifications of the DTC/DMA transfer sample program code included in USB-BASIC-F/W are listed below.
USB Pipe 1 and Pipe2 can used DTC/DMA access.
Table10-1 shows DTC/DMA Setting Specifications.

Table10-1 DTC/DMA Setting Specifications

Setting	Description
FIFO port used	D0FIFO and D1FIFO port
Transfer mode	Block transfer mode One transfer size: max packet size.
Chain transfer	Disabled
Address mode	Full address mode
Read skip	Disabled
Access bit width (MBW)	4-byte transfer: 32-bit width (when using USBA/USBA module only) 2-byte transfer: 16-bit width 1-byte transfer: 8-bit width
Transfer end	Receive direction: BRDY interrupt Transmit direction: BEMP interrupt

10.2 Notes

10.2.1 Data Reception

The area of integral multiples of the max packet size is necessary for the buffer where the received data is stored.
When the max packet size is 64 bytes, the necessary size of buffer example is shown to the receive data size below.

Table10-2 Buffer size example (max packet size is 64bytes)

Receive data size	Buffer size [bytes]
64 bytes or less	64 (max packet size)
65 bytes or more and 128 bytes or less	128 (max packet size times two)
…	…
449 bytes or more and 512 bytes or less	512 (max packet size times eight)
…	…

10.2.2 USB Pipe

USB pipe which is used by DMA/DTC transfer is only PIPE1 and PIPE2. This driver does not work properly when USB pipe except PIPE1 and PIPE2 is used

11. Additional Notes

11.1 Vendor ID

Be sure to use the user's own Vendor ID for the one to be provided in the Device Descriptor.

11.2 Compliance Test

In order to run the USB Compliance Test it is necessary to display USB device–related information on a display device such as an LCD. When the *USB_CFG_COMPLIANCE* definition in the configuration file (*r_usb_basic_config.h*) is set to *USB_CFG_ENABLE*, the USB driver calls the function (*usb_compliance_disp*) indicated below. This function should be defined within the application program, and the function should contain processing for displaying USB device–related information, etc.

Function name	:	void usb_compliance_disp(usb_compliance_t *);	
Argument	:	usb_compliance_t *	Pointer to structure for storing USB information

Note:

1. The USB driver sets the USB device–related information in an area indicated by an argument, and the *usb_compliance_disp* function is called.
2. For information on the *usb_compliance_t* structure, refer to **8.7, usb_compliance_t structure**.
3. When the *USB_CFG_COMPLIANCE* definition in *r_usb_basic_config.h* is set to *USB_CFG_ENABLE*, it is necessary to register the vendor ID and product ID in the TPL definitions for USB devices and USB hubs. For information on TPL definitions, refer to 10, Target Peripheral List (TPL) Settings.
4. For a program sample of the *usb_compliance_disp* function, see **13.1, usb_compliance_disp function**.

11.3 Hi-speed Embedded Host Electrical Test

The USB Protocol and Electrical Test Tool is required in order to run the Hi-speed embedded host electrical test. To run the test, define *USB_CFG_ELECTRICAL* in the *r_usb_basic_config.h* file as *USB_CFG_ENABLE*. For information on this definition, refer to **7.1.1, r_usb_basic_config.h**.

11.4 PIPEBUF Register Setting

Recommended values are provided for setting the BUFSIZE and BUFNMB bits of the PIPEBUF register that are supported by the USBA and USBAa modules. If it is necessary to change the settings of these bits, then change the values defined within the following files in the USB driver.

Device Class	File Name	Variable Name
Host Communication Device Class	r_usb_hcdc_driver.c	g_usb_hcdc_eptbl
Host Human I/F Device Class	r_usb_hhid_driver.c	g_usb_hhid_eptbl
Host Mass Storage Class	r_usb_hmsc_driver.c	g_usb_hmsc_eptbl
Peripheral Communication Device Class Peripheral Human I/F Device Class Periphral Mass Storage Class	r_usb_peptable.c	g_usb_eptbl

11.5 DTC transfer

Refer to "Special Note" described in the chapter "R_DTC_Open" in the application note "RX Family DTC module" (Document No. R01AN1819).

12. Creating an Application Program

This chapter explains how to create an application program using the API functions described throughout this document. Please make sure you use the API functions described here when developing your application program.

12.1 Configuration

Set each configuration file (header file) in the r_config folder to meet the specifications and requirements of your system. Please refer to chapter **7, Configuration** about setting of the configuration file.

12.2 Descriptor Creation

For USB peripheral operations, your will need to create descriptors to meet your system specifications. Register the created descriptors in the *usb_descriptor_t* function members. USB host operations do not require creation of special descriptors.

12.3 Application Program Creation

12.3.1 Include

Make sure you include the following files in your application program.

1. r_usb_basic_if.h (Inclusion is obligatory.)
2. r_usb_xxxxx_if.h (I/F file provided for the USB device class to be used)
3. Include a header file for FAT when creating the application program for Host Mass Storage Class.
4. Include any other driver-related header files that are used within the application program.

12.3.2 Initialization

1. MCU pin settings

USB input/output pin settings are necessary to use the USB controller. The following is a list of USB pins that need to be set. Set the following pins as necessary.

Table12-1 USB I/O Pin Settings for USB Peripheral Operation

Pin Name	I/O	Function
USB_VBUS	input	VBUS pin for USB communication
USB_DPUPE	output	Pull-up resistor control signal pin

Table12-2 USB I/O Pin Settings for USB Host Operation

Pin Name	I/O	Function
USB_VBUSEN	output	VBUS output enabled pin for USB communication
USB_OVRCURA	input	Overcurrent detection pin for USB communication

Note:

(1). Please refer to the corresponding MCU user's manual for the pin settings in ports used for your application program.

(2). USB_DPUPE pin is supported by RX63N/RX631 only.

(3). Make setting to DPRPD and DRPD pin as necessary when using RX63N/RX631.

2. DTC/DMA-related initialization

Call the DTC/DMA initialization fucntion when using the DTC/DMA transfer.

Tranfer	Function
DTC	R_DTC_Open
DMA	R_DMACA_Init R_DMACA_Open

Note:

(1). The setting for DTC/DMA transfer is needed when using DTC/DMA transfer. Refer to chpater **7.1.1, r_usb_basic_config.h**.

(2). You need to specify the using DMA channel number to the argument for *R_DMACA_Open* function when using DMA transfer. Be sure to specify one of the following definitions for the argument.

DMA Channel Number	Description
USB_CFG_USB0_DMA_TX	Transmission setting for USB0 module
USB_CFG_USB0_DMA_RX	Reception setting for USB0 module
USB_CFG_USB1_DMA_TX	Transmission setting for USB1 module
USB_CFG_USB1_DMA_RX	Reception setting for USB1 module

Example 1) DMA trasnmission setting of USB0 module.
R_DMACA_Open(USB_CFG_USB0_DMA_TX);
(Specify one of USB PIPE3 to USB PIPE5 for the reception USB pipe.)

Example 2) DMA reception setting of USB1 module.
R_DMACA_Open(USB_CFG_USB1_DMA_RX);
(Specify one of USB PIPE3 to USB PIPE5 for the reception USB pipe.)

Example 3) DMA transmission/reception setting of USB1 module.
R_DMACA_Open(USB_CFG_USB1_DMA_TX);
R_DMACA_Open(USB_CFG_USB1_DMA_RX);
(Don't specify USB pipe other than USB PIPE1 adn USB PIPE2.)

(3). You can use USB PIPE1 and PIPE2 when using DMA/DTC transfer. This driver does not support DMA/DTC transfer when using USB PIPE3 to PIPE5.

(4). Specify the using USB pipe number in each USB class configuration.

3. USB-related initialization

Call the *R_USB_Open* function to initialize the USB module (hardware) and USB driver software used for your application program.

12.3.3 Descriptor Creation

For USB peripheral operations please create descriptors to meet your system specifications. Refer to chapter **2.5, Descriptor** for more details about descriptors. USB host operations do not require creation of special descriptors.

12.3.4 Main routine

Please describe the main routine in the main loop format. Make sure you call the R_USB_GetEvent function in the main loop. The USB-related completed events are obtained from the return value of the *R_USB_GetEvent* function. Also make sure your application program has a routine for each return value. The routine is triggered by the corresponding return value

12.3.5 Application program description example (CPU transfer)

```
#include "r_usb_basic_if.h"
#include "r_usb_pcdc_if.h"

void            usb_peri_application( void )
{
    usb_ctrl_t   ctrl;
    usb_cfg_t    cfg;

    /* MCU pin setting */
    usb_pin_setting();

    /* Initialization processing */
    ctrl.module = USB_IP1; /* Specify the selected USB module */
    cfg.usb_mode = USB_PERI; /* Specify either USB host or USB peri */
    cfg.usb_speed = USB_HS; /* Specify the USB speed */
    cfg.usb_reg = &smp_descriptor; /* Specify the top address of the descriptor table */
```

```
        R_USB_Open( &ctrl, &cfg );

        /* main routine */
        while(1)
        {
            switch( R_USB_GetEvent( &ctrl ) )
            {
                case USB_STS_CONFIGURED:
                case USB_STS_WRITE_COMPLETE:
                    ctrl.type = USB_PCDC;
                    R_USB_Read( &ctrl, g_buf, 64 );
                    break;
                case USB_STS_READ_COMPLETE:
                    ctrl.type = USB_PCDC;
                    R_USB_Write( &ctrl, g_buf, ctrl.size );
                    break;
                default:
                    break;
            }
        }
    }
```

12.3.6 Application program description example (DMA transfer)

```
    #include "r_usb_basic_if.h"
    #include "r_usb_pcdc_if.h"

    void        usb_peri_application( void )
    {
        usb_ctrl_t   ctrl;
        usb_cfg_t    cfg;

        /* MCU pin setting */
        usb_pin_setting();

        /* DMA initialization processing */
        R_DMACA_Init();
        R_DMACA_Open(USB_CFG_USB0_DMA_TX);
        R_DMACA_Open(USB_CFG_USB0_DMA_RX);

        /* Initialization processing */
        ctrl.module = USB_IP0; /* Specify the selected USB module */
        cfg.usb_mode = USB_PERI; /* Specify either USB host or USB peri */
        cfg.usb_speed = USB_HS; /* Specify the USB speed */
        cfg.usb_reg = &smp_descriptor; /* Specify the top address of the descriptor table */
        R_USB_Open( &ctrl, &cfg );

        /* main routine */
        while(1)
        {
            switch( R_USB_GetEvent( &ctrl ) )
            {
                case USB_STS_CONFIGURED:
                case USB_STS_WRITE_COMPLETE:
                    ctrl.type = USB_PCDC;
                    R_USB_Read( &ctrl, g_buf, 64 );
                    break;
                case USB_STS_READ_COMPLETE:
                    ctrl.type = USB_PCDC;
                    R_USB_Write( &ctrl, g_buf, ctrl.size );
                    break;
                default:
                    break;
            }
        }
    }
```

12.3.7 Application program description example (DTC transfer)

```
    #include "r_usb_basic_if.h"
    #include "r_usb_pcdc_if.h"

    void        usb_peri_application( void )
    {
        usb_ctrl_t  ctrl;
        usb_cfg_t   cfg;

        /* MCU pin setting */
        usb_pin_setting();

        /* DTC initialization processing */
        R_DTC_Open();

        /* Initialization processing */
        ctrl.module = USB_IP0; /* Specify the selected USB module */
        cfg.usb_mode = USB_PERI; /* Specify either USB host or USB peri */
        cfg.usb_speed = USB_HS; /* Specify the USB speed */
        cfg.usb_reg = &smp_descriptor; /* Specify the top address of the descriptor table */
        R_USB_Open( &ctrl, &cfg );

        /* main routine */
        while(1)
        {
            switch( R_USB_GetEvent( &ctrl ) )
            {
                case USB_STS_CONFIGURED:
                case USB_STS_WRITE_COMPLETE:
                    ctrl.type = USB_PCDC;
                    R_USB_Read( &ctrl, g_buf, 64 );
                    break;
                case USB_STS_READ_COMPLETE:
                    ctrl.type = USB_PCDC;
                    R_USB_Write( &ctrl, g_buf, ctrl.size );
                    break;
                default:
                    break;
            }
        }
    }
```

13. Program Sample

13.1 usb_compliance_disp function

```
void usb_compliance_disp (usb_compliance_t *p_info)
{
    uint8_t                 disp_data[32];

    disp_data = (usb_comp_disp_t*)param;

    switch(p_info->status)
    {
        case USB_CT_ATTACH:                 /* Device Attach Detection */
            display("ATTACH ");
        break;

        case USB_CT_DETACH:                 /* Device Detach Detection */
            display("DETTACH");
        break;

        case USB_CT_TPL:                    /* TPL device connect */
            sprintf(disp_data,"TPL PID:%04x VID:%04x",p_info->pid, p_info->vid);
            display(disp_data);
        break;

        case USB_CT_NOTTPL:                 /* Not TPL device connect */
            sprintf(disp_data,"NOTPL PID:%04x VID:%04x",p_info->pid, p_info->vid);
            display(disp_data);
        break;

        case USB_CT_HUB:                    /* USB Hub connect */
            display("Hub");
        break;

        case USB_CT_NOTRESP:                /* Response Time out for Control Read Transfer */
            display("Not response");
        break;

        default:
        break;
    }
```

Note:

The display function in the above function displays character strings on a display device. It must be provided by the customer.

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	Aug 1, 2014	—	First edition issued
1.10	Dec 26, 2014	—	RX71M is added in Target Device. The multiple connecting of USB deive is supported in Host mode.
1.11	Sep 30, 2015	—	RX63N and RX631 are added in Target Device.
1.20	Sep 30, 2016	—	1. RX65N and RX651 are added in Target Device. 2. Supporting DMA transfer. 3. Supporting USB Host and Peripheral Interface Driver application note (Document No.R01AN3293EJ)
1.21	Mar 31, 2017	—	1. Supported Technical Update (Document number. TN-RX*-A172A/E) 2. The following chapters are added in this document. (1). 2.5 Descriptor (2). 3.6 How to Set the Target Peripheral List (TPL) (3). 3.7 Allocation of Device Addresses (4). 5. Return Value of R_USB_GetEvent Function (5). 6. Device Class Types (6). 7. Configuration (7). 8. Structures (8). 9. USB Class Requests (9). 11. Additional Notes (10). 13. Program Sample 3. The following chapters are deleted. "Hub Class", "non-OS Scheduler"

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.

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, HALII 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