RX Family

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

RX65N/RX651 Group
RX64M Group
RX71M Group
RX66T Group
RX72T Group
RX72M Group
RX66N Group
RX72N Group
RX671 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. RX64M Group User's Manual: Hardware (Document number. R01UH0377)
3. RX71M Group User's Manual: Hardware (Document number. R01UH0493)
4. RX65N/RX651 User's Manual: Hardware (Document number. R01UH0590)
5. RX65N/RX651-2M User's Manual: Hardware (Document number. R01UH0659)
6. RX66T User's Manual: Hardware (Document number. R01UH0749)
7. RX72T User's Manual: Hardware (Document number. R01UH0803)
8. RX72M User's Manual: Hardware (Document number. R01UH0804)
9. RX66N User's Manual: Hardware (Document number. R01UH0825)
10. RX72N User's Manual: Hardware (Document number. R01UH0824)
11. RX671 User's Manual: Hardware (Document number. R01UH0899)

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

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

Contents

1. Overview 3

2. Peripheral 7

3. Host 15

4. API Functions 22

5. Callback Function (FreeRTOS, uITRON only) 64

6. Return Value of R_USB_GetEvent Function / Retrieval of USB Completion Events   (Except for Azure RTOS) 65

7. Device Class Types (Except for Azure RTOS) 69

8. Configuration (r_usb_basic_config.h) 70

9. Structures 75

10. USB Class Requests (Except for Azure RTOS) 79

11. DTC/DMA Transfer.. 86

12. Additional Notes 88

13. Creating an Application Program 96

14. Program Sample 100

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 Peripheral.
- 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 transfer)
- This driver supports RTOS version (hereinafter called "RTOS") and Non-OS version (hereinafter called "Non-OS"). RTOS uses the realtime OS. Non-OS does not use the real time OS.
- The RTOS USB driver supports FreeRTOS, uITRON(RI600V4) and Azure RTOS (USBX).

<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)	RX65N/RX651	USBb module
	RX64M	USBb module
	RX71M	USBb module
	RX66T	USBb module
	RX72T	USBb module
	RX72M	USBb module
	RX66N	USBb module
	RX72N	USBb module
	RX671	USBb module
USB1 module (Start address: 0xA0200/0xD0400)	RX64M	USBA module
	RX71M	USBAa module
	RX671	USBb 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. Multiple configurations are not supported.

4. Multiple interfaces are not supported.

5. The USB host and USB peripheral modes cannot operate at the same time.

6. 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.

7. 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.

8. In the case of Vendor class, the user can not use the USB Hub.

9. This driver does not support the CPU transfer using D0FIFO/D1FIFO register.

10. This driver does not support FreeRTOS and uITRON when using GCC or IAR compiler.

11. The user can use Azure RTOS API and USBX API in the application program when using Azure RTOS.

12. This driver supports only RX65N and RX72N when using Azure RTOS(USBX PCDC).

13. This driver supports only RX65N, RX72N(RX66N) and RX671 when using Azure RTOS(USBX HMSC).

14. This driver supports only RX65N, RX72N and RX671 when using Azure RTOS(USBX HCDC)

15. USB Hub is not supported when using Azure RTOS.

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 for USB-BASIC-FW
HDCD	:	Host Device Dlass Driver (device driver and USB class driver)
HUBCD	:	Hub Class Driver
H/W	:	Renesas USB device
IDE	:	Integrated Development Environment
MGR	:	Peripheral Device State Manager for HCD
Non-OS	:	USB Driver for OS-less
PBC	:	Peripheral Battery Charging control
PCD	:	Peripheral Control Driver of USB-BASIC-FW
PDCD	:	Peripheral Device Class Driver (Device driver and USB class driver)
RTOS	:	USB Driver for FreeRTOS and uITRON
USB-BASIC-FW	:	USB Basic Host and Peripheral Driver
Scheduler	:	Used to schedule functions, like a simplified OS.
Task	:	Processing unit

1.4　USB-BASIC-FW 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.

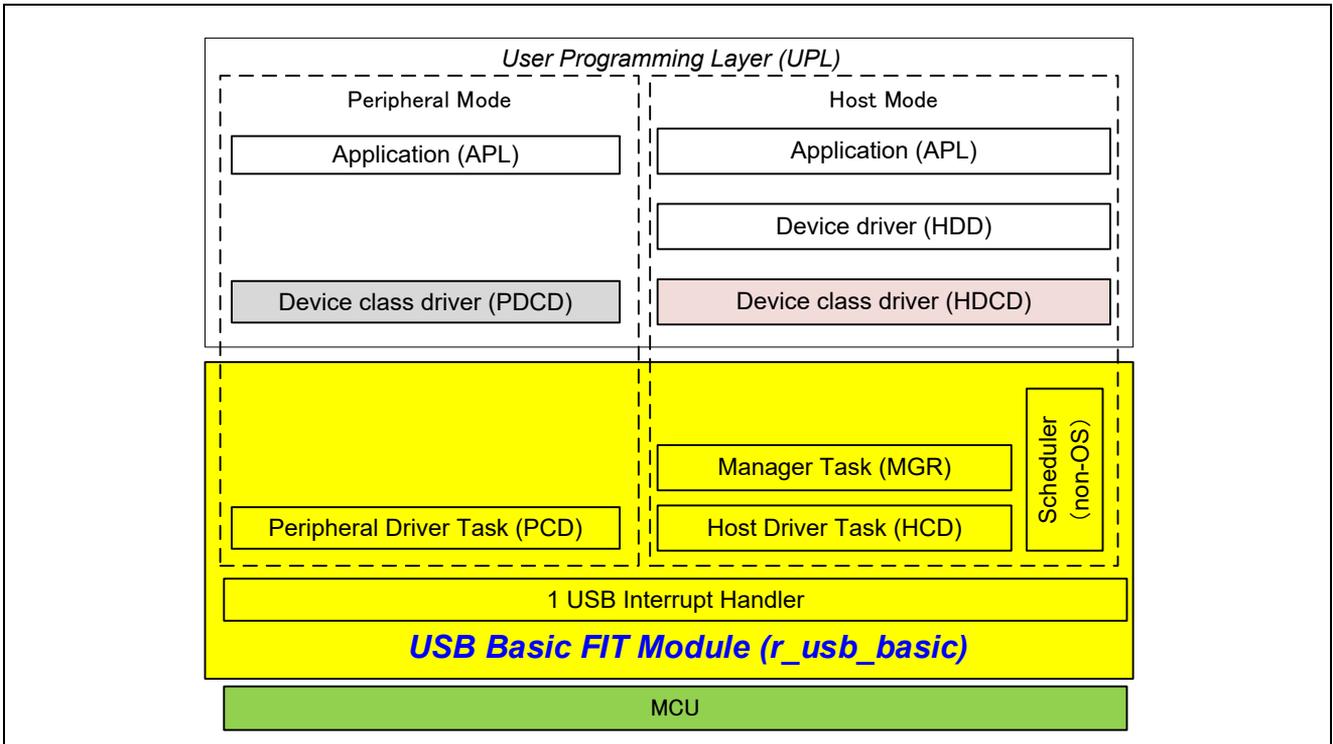## 1.5   Software Configuration

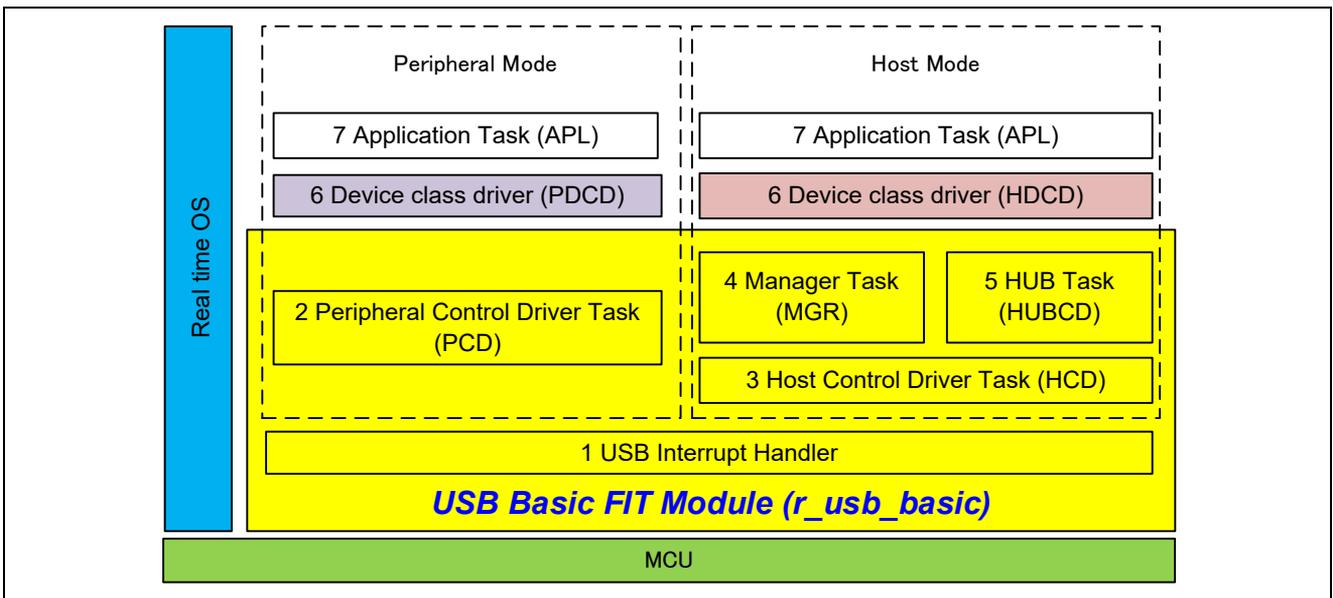Figure 1-1 USB Software Configuration of USB-BASIC-FW (Non-OS)

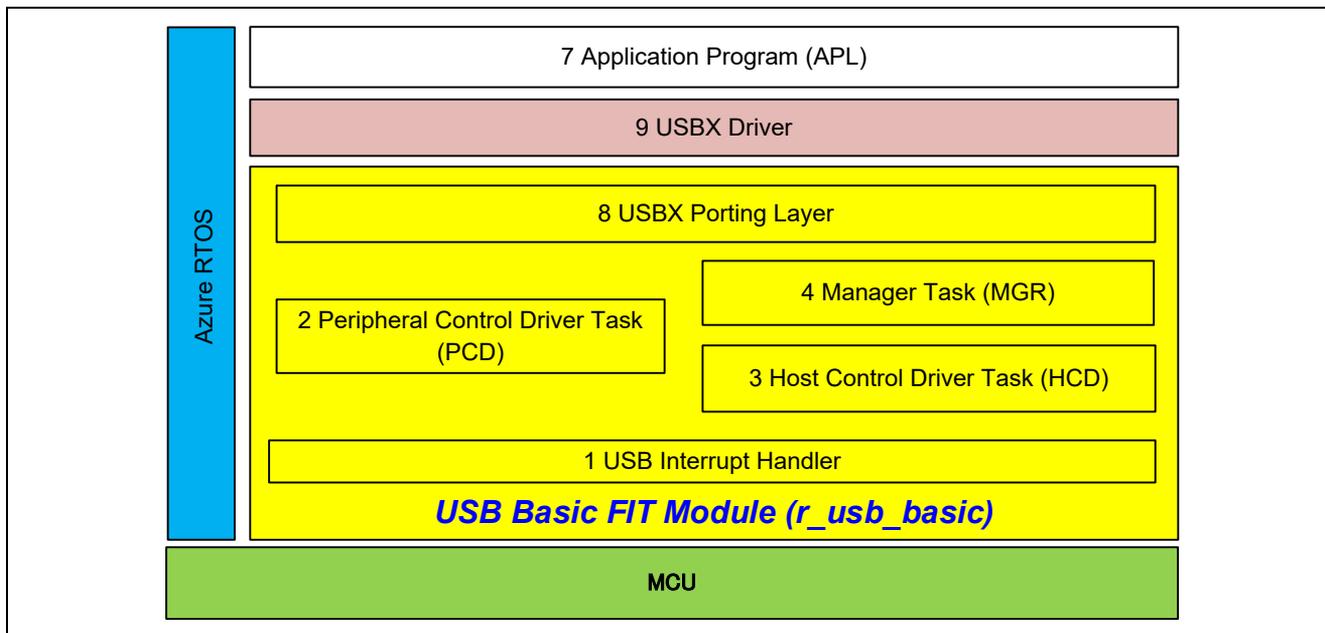Figure 1-2   USB Software Configuration of USB-BASIC-FW (FreeRTOS, uITRON)

Figure 1-3   USB Software Configuration of USB-BASIC-FW (Azure RTOS (USBX))

Table 1-1   Software function overview

No	Module Name	Function
1	USB Interrupt Handler	USB interrupt handler (USB packet transmit/receive end and special signal detection)
2	Peripheral Control Driver (PCD)	Hardware control in peripheral mode Peripheral transaction management
3	Host control driver (HCD)	Hardware control in host mode Host transaction management
4	Host Manager (MGR)	Device state management Enumeration HCD/HUBCD control message determination
5	HUB Driver (HUBCD)	HUB down port device state management HUB down port enumeration
6	Device Class Driver (PCDC, HCDC)	USB Device Class Driver
7	Application Program	USB Application Program
8	USBX Porting Layer	Porting Layer for USBX Driver
9	USBX Driver	USBX Device Class Driver

1.6   Scheduler Function and Tasks

This driver (Non-OS) manages requests generated by tasks and hardware according to their relative priority by using scheduler. When multiple requests are generated to multiple tasks with the same priority, they are executed using a FIFO configuration. Requests between tasks are implemented by transmitting and receiving messages.

1.7   Pin Setting

To use the USB FIT module, input/output signals of the peripheral function has to be allocated to pins with the multi-function pin controller (MPC). 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.

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 10, 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   Operating Confirmation Environment

Item	Contents
C compiler	Renesas Electronics C/C++ compiler for RX Family V.3.07.00 (The option "-lang=C99" is added to the default setting of IDE)
	GCC for Renesas RX 8.3.0.202411 (The option "-std=gnu99" is added to the default setting of IDE)
	IAR C/C++ Compiler for Renesas RX version 5.10.1
Real-Time OS	FreeRTOS V.10.0.0
	RI600V4
	Azure RTOS (USBX) 6.1.12
Endian	Little Endian, Big Endian
USB Driver Revision Number	Rev.1.44
Using Board	Renesas Starter Kits for RX64M
	Renesas Starter Kits for RX71M
	Renesas Starter Kits for RX65N, Renesas Starter Kits for RX65N-2MB
	Renesas Starter Kits for RX72T
	Renesas Starter Kits for RX72M
	Renesas Starter Kits for RX72N
	Renesas Starter Kits for RX671
Host Environment	The operation of this USB Driver module connected to the following OSes has been confirmed. 1. 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
RX64M RX71M	USBI0 Interrupt (Vector number: 189, Interrupt source number : 62, Software Configurable Interrupt B)
	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 RX72M RX72N RX66N	USBI0 Interrupt (Vector number: 185, Interrupt source number : 62, Software Configurable Interrupt B)
	USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35)
	USBR0 Interrupt (Vector number:90)
RX66T RX72T	USBI0 Interrupt (Vector number: 174) / USBR0 Interrupt (Vector number: 90)
	USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35)
RX671	USBI0 Interrupt (Vector number: 181, Interrupt source number : 62, Software Configurable Interrupt B)
	USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35)
	USBR0 Interrupt (Vector number:90)
	USBI1 Interrupt (Vector number: 182, Interrupt source number : 63, Software Configurable Interrupt B)
	USB D0FIFO1 Interrupt (Vector number: 36) / USB D1FIFO1 Interrupt (Vector number: 37)

2.2.5   Timer

This driver (RTOS) uses a timer (CMT) in RX MCU. If a timer is to be used in the user system, use a timer other than a timer is used by this driver.

2.2.6   Header Files

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

2.2.7   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*.

2.2.8   Compile Setting

For compile settings, refer to chapter 8, Configuration.

2.2.9   ROM / RAM Size

The follows show ROM/RAM size of this driver.

1.   CC-RX (Optimization Level: Default)

   (1).   Non-OS

	Checks arguments	Does not check arguments
ROM size	21.7K bytes (Note 3)	21.3K bytes (Note 4)
RAM size	5.5K bytes	5.5K bytes

   (2).   RTOS

   a.   FreeRTOS

	Checks arguments	Does not check arguments
ROM size	35.1K bytes (Note 3)	34.7K bytes (Note 4)
RAM size	21.0K bytes	21.0K bytes

   b.   RI600V4

	Checks arguments	Does not check arguments

RENESAS

ROM size	36.6K bytes (Note 3)	36.2K bytes (Note 4)
RAM size	11.2K bytes	11.2K bytes

2. GCC (Optimization Level: -O2)

	Checks arguments	Does not check arguments
ROM size	26.6K bytes (Note 3)	26.1K bytes (Note 4)
RAM size	5.3K bytes	5.3K bytes

3. IAR (Optimiza Level: Medium)

	Checks arguments	Does not check arguments
ROM size	21.1K bytes (Note 3)	20.6K bytes (Note 4)
RAM size	4.0K bytes	4.0K bytes

Note:

1. ROM/RAM size for BSP and the vendor class is included in the above size.

2. The above is the size when specifying RX V2 core option.

3. 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.

4. 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.

5. The result of RTOS includes the ROM/RAM size of the real-time OS.

6. Please refer to each device class documentation for ROM/RAM size when using Azure RTOS.

2.2.10 Argument

For the structure used in the argument of API function, refer to chapter 9, Structures.

2.2.11 "for", "while" and "do while" statements.

In FIT module, when using "for", "while" and "do while" statements (loop processing) in register reflection waiting processing, etc., write comments with "WAIT_LOOP" as a keyword for these loop processing. Also, write in the FIT documentation that "WAIT_LOOP" is written as a comment in these loop processes.

2.2.12 Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends the method using the Smart Configurator described in (1) or (3) below. However, the Smart Configurator only supports some RX devices. Please use the methods of (2) or (4) for RX devices that are not supported by the Smart Configurator.

(1) Adding the FIT module to your project using the Smart Configurator in e^2 studio

By using the Smart Configurator in e^2 studio, the FIT module is automatically added to your project. Refer to "Renesas e^2 studio Smart Configurator User Guide (R20AN0451)" for details.

(2) Adding the FIT module to your project using the FIT Configurator in e^2 studio

By using the FIT Configurator in e^2 studio, the FIT module is automatically added to your project. Refer to "Adding Firmware Integration Technology Modules to Projects (R01AN1723)" for details.

(3) Adding the FIT module to your project using the Smart Configurator in CS+

By using the Smart Configurator Standalone version in CS+, the FIT module is automatically added to your project. Refer to "Renesas e^2 studio Smart Configurator User Guide (R20AN0451)" for details.

(4)   Adding the FIT module to your project on CS+

In CS+, please manually add the FIT module to your project. Refer to "Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)" for details.

2.3 API (Application Programming Interface)

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

Note:
Please refer to the documentation for Azure RTOS and USBX when using Azure RTOS.

2.4 Class Request (Except for Azure RTOS)

For the processing method when this driver receives the class request, refer to chapter 10, USB Class Requests.

2.5 Descriptor (Except for Azure RTOS)

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 (*pp_string)*. Refer to chapter 9.4, usb_descriptor_t structure for more details concerning the *usb_descriptor_t* structure.

4. Set the number of the string descriptor which set in the string descriptor table to *usb_descriptor_t* structure member (*num_string*). In the case of the above example, the value 3 is set to the member (*num_string*).

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 9.4, usb_descriptor_t structure.

2.6　　Peripheral Battery Charging (PBC) (Except for Azure RTOS)

This driver supports PBC.

PBC is a H / W control program to operate the target device as a Portable Device for Battery Charging defined by the USB Battery Charging Specification (Revision 1.2).

You can get the result of whether USB Host is the SDP or CPD by calling *R_USB_GetInformation* function. For *R_USB_GetInformation* function, refer to chapter 4.13.

Note:

　The following MCU does not support the PBC.

1. RX651/RX65N

2. RX66T

3. RX72T

4. RX72M

5. RX72N

6. RX66N

7. RX71M (USB0 module)

8. RX64M (USB0 module)

9. RX671

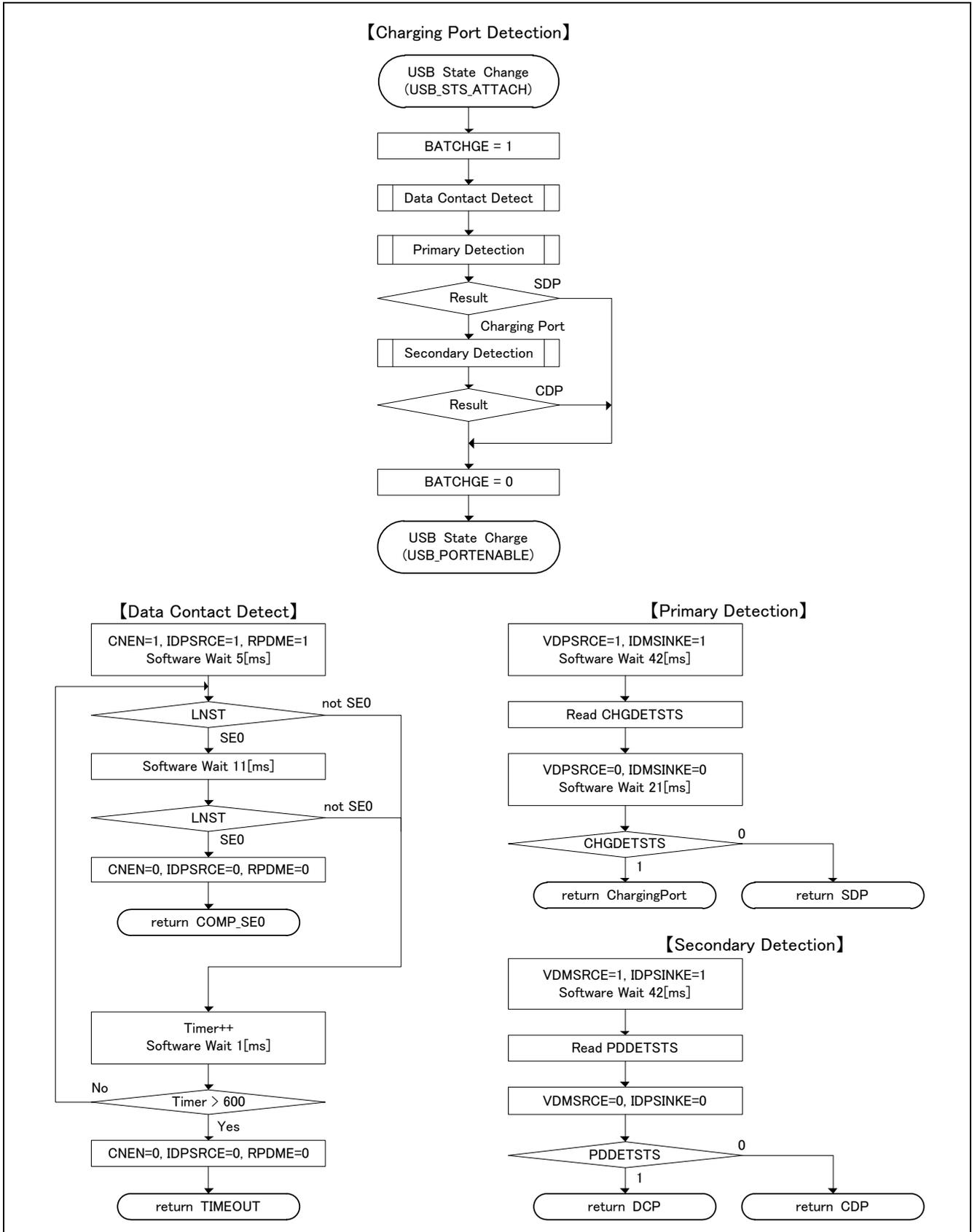10. The processing flow of PBC is shown in Figure 2-1.

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.

GET_DESCRIPTOR（Device Descriptor）
SET_ADDRESS
GET_DESCRIPTOR（Configuration Descriptor）
SET_CONFIGURATION
SET_FEATURE (HID only)
CLEAR_FEATURE (HID only)

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   Operating Confirmation Environment

Item	Contents
C compiler	Renesas Electronics C/C++ compiler for RX Family V.3.07.00 (The option "-lang=C99" is added to the default setting of IDE)
	GCC for Renesas RX 8.3.0.202411 (The option "-std=gnu99" is added to the default setting of IDE)
	IAR C/C++ Compiler for Renesas RX version 5.10.1
Real-Time OS	FreeRTOS V.10.0.0
	RI600V4
	Azure RTOS 6.1.12
Endian	Little Endian, Big Endian
USB Driver Revision Number	Rev.1.44
Using Board	Renesas Starter Kits for RX64M
	Renesas Starter Kits for RX71M
	Renesas Starter Kits for RX65N, Renesas Starter Kits for RX65N-2MB
	Renesas Starter Kits for RX72T
	Renesas Starter Kits for RX72M
	Renesas Starter Kits for RX72N
	Renesas Starter Kits for RX671

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
RX64M RX71M	USBI0 Interrupt (Vector number: 189, Interrupt source number : 62, Software Configurable Interrupt B)
	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 RX72M RX72N RX66N	USBI0 Interrupt (Vector number: 185, Interrupt source number : 62, Software Configurable Interrupt B) USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35) USBR0 Interrupt (Vector number:90)
RX66T RX72T	USBI0 Interrupt (Vector number: 174) / USBR0 Interrupt (Vector number: 90) USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35)
RX671	USBI0 Interrupt (Vector number: 185, Interrupt source number : 62, Software Configurable Interrupt B) USB D0FIFO0 Interrupt (Vector number: 34) / USB D1FIFO0 Interrupt (Vector number: 35) USBR0 Interrupt (Vector number:90) USBI1 Interrupt (Vector number: 182, Interrupt source number : 63, Software Configurable Interrupt B) USB D0FIFO1 Interrupt (Vector number: 36) / USB D1FIFO1 Interrupt (Vector number: 37)

3.3.5   Timer

This driver (FreeRTOS) uses a timer (CMT) in RX MCU. If a timer is to be used in the user system, use a timer other than a timer is used by this driver.

3.3.6   Header Files

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

3.3.7   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*.

3.3.8   Compile Setting

For compile settings, refer to chapter 8, Configuration..

3.3.9   ROM / RAM Size

The follows show ROM/RAM size of this driver.

1.   CC-RX (Optimization Level: Default)

   (1).   Non-OS

	Checks arguments	Does not check arguments
ROM size	36.4K bytes (Note 3)	36.0K bytes (Note 4)
RAM size	13.2K bytes	13.2K bytes

   (2).   RTOS

   a. FreeRTOS

	Checks arguments	Does not check arguments
ROM size	47.3K bytes (Note 3)	46.8K bytes (Note 4)
RAM size	34.6K bytes	34.6K bytes

   b. RI600V4

	Checks arguments	Does not check arguments
ROM size	49.4K bytes (Note 3)	48.9K bytes (Note 4)
RAM size	16.8K bytes	16.8K bytes

2.   GCC (Optimization Level: -O2)

RENESAS

	Checks arguments	Does not check arguments
ROM size	42.4K bytes (Note 3)	41.8K bytes (Note 4)
RAM size	13.0K bytes	13.0K bytes

3.   IAR (Optimization Level: Medium)

	Checks arguments	Does not check arguments
ROM size	35.2K bytes (Note 3)	34.6K bytes (Note 4)
RAM size	11.7K bytes	11.7K bytes

Note:

1.   ROM/RAM size for BSP and the vendor class is included in the above size.

2.   The above is the size when specifying RX V2 core option.

3.   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.

4.   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.

5.   The result of RTOS includes the ROM/RAM size of the real-time OS.

6.   Please refer to each device class documentation for ROM/RAM size when using Azure RTOS.

3.3.10   Argument

For the structure used in the argument of API function, refer to chapter 9, Structures.

3.3.11   "for", "while" and "do while" statements.

In FIT module, when using "for", "while" and "do while" statements (loop processing) in register reflection waiting processing, etc., write comments with "WAIT_LOOP" as a keyword for these loop processing. Also, write in the FIT documentation that "WAIT_LOOP" is written as a comment in these loop processes.

3.3.12   Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends the method using the Smart Configurator described in (1) or (3) below. However, the Smart Configurator only supports some RX devices. Please use the methods of (2) or (4) for RX devices that are not supported by the Smart Configurator.

(1)   Adding the FIT module to your project using "Smart Configurator" on e^2 studio

By using the Smart Configurator in e^2 studio, the FIT module is automatically added to your project. Refer to "Renesas e^2 studio Smart Configurator User Guide (R20AN0451)" for details.

(2)   Adding the FIT module to your project using the FIT Configurator in e^2 studio

By using the FIT Configurator in e^2 studio, the FIT module is automatically added to your project. Refer to "Adding Firmware Integration Technology Modules to Projects (R01AN1723)" for details.

(3)   Adding the FIT module to your project using the Smart Configurator in CS+

By using the Smart Configurator Standalone version in CS+, the FIT module is automatically added to your project. Refer to "Renesas e^2 studio Smart Configurator User Guide (R20AN0451)" for details.

(4)   Adding the FIT module to your project on CS+

In CS+, please manually add the FIT module to your project. Refer to "Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)" for details.

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 10, USB Class Requests.

3.6   How to Set the Target Peripheral List (TPL)   (Except for Aure RTOS)

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          0x0011, 0x0022, 0x0033, 0x0044, 0x0055, 0x0066
                               VID     PID     VID     PID     VID     PID

                               USB device 1    USB device 2    USB device 3


#define       USB_HUB_TPL      0x1111, 0x2222, 0x3333, 0x4444
                               VID     PID     VID     PID

                               USB Hub1        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 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 8.

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)　(Except for Aure RTOS)

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:

 The following MCU does not support the HBC.

   1. RX651/RX65N

   2. RX66T

   3. RX72T

   4. RX72M

   5. RX72N

   6. RX66N

   7. RX71M (USB0 module)

   8. RX64M (USB0 module)

   9. RX671

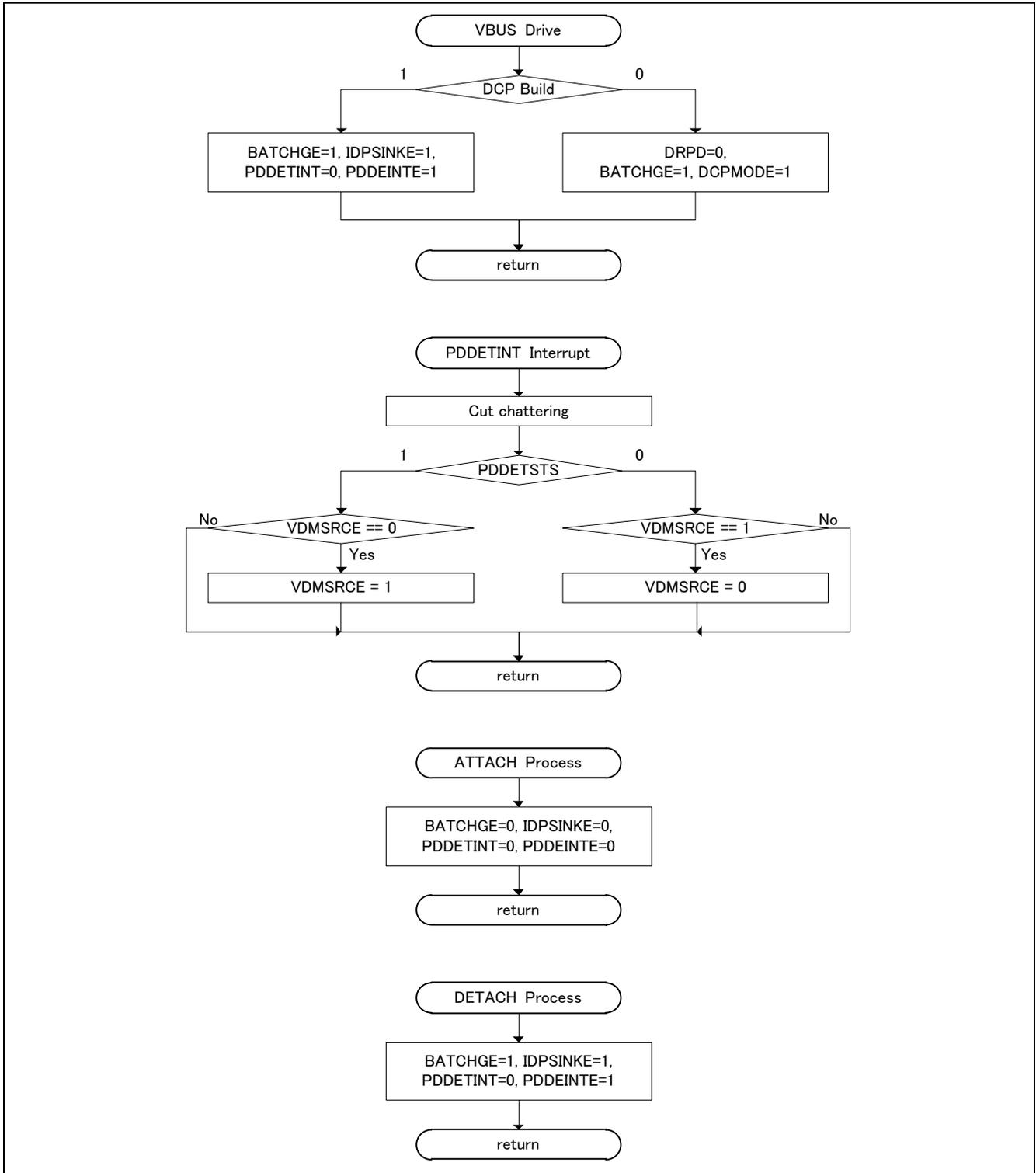The processing flow of HBC is shown Figure 3-1.

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 (Non-OS only)
R_USB_Callback()	Register a callback function (RTOS only)
R_USB_VbusSetting()	Sets VBUS supply start/stop.
R_USB_PullUp()	Pull-up enable/disable setting of D+/D- line.
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.

3. When *USB_CFG_DISABLE* is specified to *USB_CFG_PARAM_CHECKING* definition, the return value *USB_ERR_PARA* is not returned since this driver does not check the argument. Refer to chapter 8, Configuration for *USB_CFG_PARAM_CHECKING* definition.

4. Please use Azure RTOS API and USBX API when using Azure RTOS. Please refer to Azure RTOS and USBX documentation for these APIs. Of the above APIs, use only the *R_USB_Open* function and *R_USB_Close* function. Do not use other than *R_USB_Open* and *R_USB_Close* function.

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 9.1, usb_ctrl_t structure, and for the *usb_cfg_t* structure, see chapter 9.3, usb_cfg_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.   Assign the device class type (see chapter 7, Device Class Types) to the member *(type)* of the *usb_ctrl_t* structure. Does not assign *USB_HCDCC* and *USB_PCDCC* to this member (*type*). If *USB_HCDCC* and *USB_PCDCC* is assinged, then *USB_ERR_PARA* will be returned.

5.   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.

6.   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.

7.   Assign a pointer to the usb_descriptor_t structure to the member (*p_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 (*p_usb_reg*) is ignored.

8.   If 0 (zero) is assigned to one of the arguments, *USB_ERR_PARA* will be the return value.

9.   Do not call this API in the multiple tasks. (RTOS)

10.   Do not call this API in the following function.

   (1). Interrupt function.

   (2). Callback function regsitered by *R_USB_Callback* function.

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;
    ctrl.type = USB_HCDC;
    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;
    ctrl.type = USB_PCDC;
    cfg.usb_mode = USB_PERI;
    cfg.usb_speed = USB_HS;
    cfg.p_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.

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.

3.   If *USB_NULL* is assigned to the argument (*p_ctrl*), then *USB_ERR_PARA* will be the return value.

4.   Do not call this API in the multiple tasks. (RTOS)

5.   Do not call this API in the following function.

   (1).   Interrupt function.

   (2).   Callback function regsitered by *R_USB_Callback* function。

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

    (1). Non-OS

    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). RTOS

    Requests USB data read (bulk/interrupt transfer).
    The read data is stored in the area specified by argument (*p_buf*).
    It is possible to check for the completion of a data read based on an argument (*USB_STS_READ_COMPLETE* in
    the member (*event*) of the *usb_ctrl_t* structure) to the callback function registered in the USB driver.
    After confirming the argument to the callback function registered in the USB driver
    (*USB_STS_READ_COMPLETE* in the member (*event*) of the *usb_ctrl_t* structure), reference the size member of
    the *usb_ctrl_t* structure to ascertain the size of the data from the completed read.

2. Control data transfer

    Refer to chapter 10, USB Class Requests for details.

Note

1. Please specify a multiple of MaxPacketSize to the 3rd argument (*size*).

2. This API only performs data read request processing. An application program does not wait for data read
    completion by using this API.

3. 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.

4. 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. (Non-OS)

5. If the data that has been read is not n times the maximum packet size and does not satisfy the read request size,
    the USB driver will assume that the data transfer is still in progress, so it will not call the callback function that
    provides notification that data reception is complete. (RTOS)

6. Before calling this API, assign the device class type (see chapter 7, 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.

7. 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.

8. Do not assign a pointer to the auto variable (stack) area to the second argument (*p_buf*).

9. Specify the following address for the 2nd argument (*p_buf*) when using DMA/DTC transfer.

  (1). When using USBA/USBAa module

  Start address of the buffer area aligned on 4-byte boundary.

  (2). When using other than USBA/USBa module

  Start address of the buffer area aligned on 2-byte boundary.

10. Allocate the area of the following size when using DTC/DMA transer.

  (1). When *USB_CFG_CNTMDON* is specified for *USB_CFG_CNTMD* definition in *r_usb_basic_config.h* (When using USBA/USBAa module.)

  Allocate the area more than n times FIFO buffer size. For FIFO buffer size, refer to the chapter 12.4, Change of PIPEBUF Register

  (2). When *USB_CFG_CNTMDOFF* is specified for *USB_CFG_CNTMD* definition in *r_usb_basic_config.h*.

  Allocate the area n times the max packet size.

11. 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*).

12. If 0 (zero) is assigned to one of the arguments, *USB_ERR_PARA* will be the return value.

13. 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)* 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)*, first check the *USB_STS_READ_COMPLETE* return value from the *R_USB_GetEvent* function, and then call the *R_USB_Read* function.(Non-OS)

14. 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. (Non-OS)

15. In Vendor Class, use the *R_USB_PipeRead* function.

16. If this API is called after assigning *USB_PCDCC, USB_HMSC, USB_PMSC, USB_HVND* or *USB_PVND* to the member *(type)* of the *usb_crtl_t* structure, then *USB_ERR_PARA* will be the return value.

17. In Host Mass Storage Class, to access storage media, use the FAT (File Allocation Table) API rather than this API.

18. 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.

19. Do not call this API in the following function.

  (1). Interrupt function.

  (2). Callback function regsitered by *R_USB_Callback* function.

Example

1.   Non-OS

```
uint8_t  g_buf[512];
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;
                            :
        }
    }
}
```

2.　RTOS

```c
uint8_t g_buf[512];
/* Callback function */
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
    USB_APL_SND_MSG(task_id, (usb_msg_t *)p_ctrl);
}

void      usb_application_task( void )
{
    usb_ctrl_t      ctrl;
    usb_ctrl_t *p_mess;
                :
    while(1)
    {
            USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
            ctrl = *p_mess;
            switch (ctrl.event)
                        :
              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

   (1).   Non-OS

      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).   RTOS

      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)*.
      It is possible to check for the completion of a data write based on an argument (*USB_STS_WRITE_COMPLETE* in the member (*event*) of the *usb_ctrl_t* structure) to the callback function registered in the USB driver.
      To request the transmission of a NULL packet, assign *USB_NULL*(0) to the third argument (*size*).

2.   Control data transfer

      Refer to chapter 10, 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.

3.   Before calling this API, assign the device class type (see chapter 7, 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 specify a pointer to the auto variable (stack) area for the second argument (*p_buf*).

6.   Specify the following address for the 2nd argument (*p_buf*) when using DMA/DTC transfer.

   (1).   When using USBA/USBAa module

Start address of the buffer area aligned on 4-byte boundary.

(2).   When using other than USBA/USBa module

Start address of the buffer area aligned on 2-byte boundary.

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 *(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 (*address*), first check the *USB_STS_WRITE_COMPLETE* return value from the *R_USB_GetEvent* function, and then call the *R_USB_Write* function. (Non-OS)

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. (Non-OS)

11.   In Vendor Class, use the *R_USB_PipeWrite* function.

12.   If this API is called after assigning *USB_HCDCC*, *USB_HMSC*, *USB_PMSC, USB_HVND* or USB_PVND 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.

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.

15.   Do not call this API in the following function.

(1).   Interrupt function.

(2).   Callback function regsitered by *R_USB_Callback* function.

Example

1. Non-OS

```
uint8_t g_buf[512];
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, 512);
                    :
            break;
            case USB_STS_WRITE_COMPLETE:
                    :
            break;
                    :
        }
    }
}
```

2. RTOS

```
uint8_t    g_buf[512];
/* Callback function */
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
    USB_APL_SND_MSG(task_id, (usb_msg_t *)p_ctrl);
}

void      usb_application_task( void )
{
    usb_ctrl_t      ctrl;
    usb_ctrl_t      *p_mess;
                 :
    while(1)
    {
            USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
            ctrl = *p_mess;
            switch (ctrl.event)
            {
                         :
              case USB_STS_READ_COMPLETE:
                         :
                  ctrl.module = USB_IP0;
                  ctrl.address = adr;
                  ctrl.type = USB_HCDC;
                  R_USB_Write(&ctrl, g_buf, 512);
                         :
              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 *USB_NULL* is assigned to the argument (*p_ctrl*), then *USB_ERR_PARA* will be the return value.

4. If something other than *USB_READ* or *USB_WRITE* is assigned to the 2nd argument (*type*), then *USB_ERR_PARA* will be the return value. When *USB_NULL* is set to the 2nd argument (*type*), this driver operates the same processing as when *USB_READ* is set.

5. If *USB_HCDCC* is assigned to the member (*type*) and *USB_WRITE* is assigned to the 2nd argment (*type*), then *USB_ERR_PARA* will be the return value.

6. If *USB_PCDCC* is assigned to the member (*type*) and *USB_READ* is assigned to the 2nd argment (*type*), then *USB_ERR_PARA* will be the return value.

7. In USB Host mode, *USB_ERR_NG* will be the return value when this API can not stop the data read/write request.

8. When the *R_USB_GetEvent* function is called after a data read/write stopping has been completed, the return value *USB_STS_READ_COMPLETE/USB_STS_WRITE_COMPLETE* is returned. (Non-OS)

9. USB driver set *USB_STS_READ_COMPLETE* or *USB_STS_WRITE_COMPLTE* to the argument (the member (*event*) of the *usb_ctrl_t* structure) in the callback function registered in the USB driver when a data read/write stopping has been completed. (RTOS)

10. If this API is called after assigning *USB_HMSC, USB_PMSC, USB_HVND* or *USB_PVND* to the member *(type)* of the *usb_crtl_t* structure, then *USB_ERR_PARA* will be the return value.

11. In Vendor Class, use the *R_USB_PipeStop* function.

12. Do not use this API for the Host Mass Storage Class.

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.

16. Do not call this API in the following function.

    (1). Interrupt function.

    (2). Callback function regsitered by *R_USB_Callback* function.

Example

1. Non-OS

```
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;
                        :
          }
      }
  }
```

RENESAS

2. RTOS

```
/* Callback function */
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
    USB_APL_SND_MSG(task_id, (usb_msg_t *)p_ctrl);
}

void        usb_application_task( void )
{
    usb_ctrl_t      ctrl;
    usb_ctrl_t      *p_mess;
                :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
        {
                    :
          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	During a suspend request to the specified USB module, or when the USB module is already in the suspended state
USB_ERR_NG	Other error

Description

1. Non-OS

   This function sends a SUSPEND signal from the USB module assigned to the member *(module)* of the *usb_crtl_t* structure. After the suspend request is completed, confirm the operation with the return value (*USB_STS_SUSPEND*) of the *R_USB_GetEvent* function.

2. RTOS

   This function sends a SUSPEND signal from the USB module assigned to the member *(module)* of the *usb_ctrl_t* structure.
   It is possible to check for the completion of the suspend request based on an argument (*USB_STS_SUSPEND* in the member *(event)* of the *usb_ctrl_t* structure) to the callback function registered in the USB driver.

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. If *USB_NULL* is assigned to the argument (*p_ctrl*), then *USB_ERR_PARA* will be the return value.

7. When this API is called in the state of other than the configured or the suspend state, *USB_ERR_NG* is returned.

8. Do not call this API in the following function.

   (1). Interrupt function.

   (2). Callback function regsitered by *R_USB_Callback* function.

Example

1.　Non-OS

```
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;
                    :
        }
    }
}
```

2.　RTOS

```
/* Callback function */
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
    USB_APL_SND_MSG(id, (usb_msg_t *)p_ctrl);
}

void usb_application_task( void )
{
    usb_ctrl_t      ctrl;
    usb_ctrl_t      *p_mess;
            :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
        {
                    :
                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	During a resume request processing
USB_ERR_NOT_SUSPEND	USB device is not in the SUSPEND state.
USB_ERR_NG	USB device is not in a state that can request the remote wakeup. (USB peripheral mode only)

Description

1.   Non-OS

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

2.   RTOS

This function sends a RESUME signal from the USB module assigned to the member *(module)* of the *usb_ctrl_t* structure.
It is possible to check for the completion of the resume request based on an argument (*USB_STS_RESUME* in the member *(event)* of the *usb_ctrl_t* structure) to the callback function registered in the USB driver.

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.   Please call this API after calling the *R_USB_Open* function (and before calling the *R_USB_Close* function).

3.   In USB Peripheral mode, this API can be used for sending RemoteWakeup signal only when receiving *SetFeature* commnad which *DEVICE_REMOTE_WAKEUP* is specified to *Feature Selector*. If this API is called before receiving the *SetFeature* command, then *USB_ERR_NG* will be the return value.

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.   If *USB_NULL* is assigned to the argument (*p_ctrl*), then *USB_ERR_PARA* will be the return value.

6.   This API can be called when the USB device is in the suspend state. When the API is called in any other state, *USB_ERR_NOT_SUSPEND* is returned.

7.   Do not call this API in the following function.

    (1).   Interrupt function.

    (2).   Callback function regsitered by *R_USB_Callback* function.

Example

1. Non-OS

(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 USB Peripheral mode

```
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;
                        :
         }
    }
}
```

2. RTOS

(1). In the case of USB Host mode

```
/* Callback function */
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
```

```
        USB_APL_SND_MSG(task_id, (usb_msg_t *)p_ctrl);
    }

    void        usb_application_task( void )
    {
        usb_ctrl_t      ctrl;
        usb_ctrl_t      *p_mess;
                :
        while(1)
        {
            USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
            ctrl = *p_mess;
            switch (ctrl.event)
            {
                    :
                    ctrl.module = USB_IP0;
                    R_USB_Resume( &ctrl );
                        :
                break;
                case USB_STS_RESUME:
                            :
                break;
                        :
            }
        }
    }
```

(2).　In the case of USB Peripheral mode

```
    void        usb_peri_application( void )
    {
        usb_ctrl_t      ctrl;
        usb_ctrl_t      *p_mess;
                :
        while(1)
        {
            USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
            ctrl = *p_mess;
            switch (ctrl.event)
            {
                        :
                    R_USB_Resume(&ctrl);
                        :
                break;
                case USB_STS_RESUME:
                        :
                break;
                        :
            }
        }
    }
```

4.9　　R_USB_GetEvent

Get completed USB-related events (Non-OS only)

Format

usb_status_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. Please call this API after calling the *R_USB_Open* function (and before calling the *R_USB_Close* function).

2. Refer to chapter 6, Return Value of R_USB_GetEvent Function / Retrieval of USB Completion Events " 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.

4. Please call this API in the main loop of the user application program.

5. Do not call this API in the interrupt function.

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_Callback

Register a callback function to be called upon completion of a USB-related event. (RTOS only)

Format

    void   R_USB_Callback(usb_callback_t *p_callback)

Arguments

    p_callback   Pointer to the callback function

Return Value

    --

Description

    This function registers a callback function to be called when a USB-related event has completed.

    When a USB-related event has completed, the USB driver will call the callback function that has been registered using this API.

Note

1. Call this API after calling the R_USB_Open function (and before calling the R_USB_Close function).

2. For details regarding the USB event values that are specified as arguments to this API, see chapter 6, Return Value of R_USB_GetEvent Function / Retrieval of USB Completion Events.

3. For information regarding callback functions, see chapter 5, Callback Function.

4. Do not call this API in the interrupt function.

Example

```
void usb_apl_callback (usb_ctrl_t *p_ctrl)
{
    USB_APL_SND_MSG(USB_APL_MBX, (usb_msg_t *)p_ctrl);
}

void usb_application_task(void)
{
    usb_ctrl_t   ctrl;
    usb_ctrl_t   *p_mess;
    usb_cfg_t    cfg;

    usb_pin_setting(); /* USB MCU pin setting */

    ctrl.module      = USE_USBIP;
    ctrl.type        = USB_PCDC;
    cfg.usb_speed    = USB_SUPPORT_SPEED;    /* USB_HS/USB_FS */
    cfg.p_usb_reg    = (usb_descriptor_t *)&usb_descriptor;
    R_USB_Open(&ctrl, &cfg);                 /* Initializes the USB module */

    R_USB_Callback(usb_apl_callback);

    while (1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);

        ctrl = *p_mess;

        switch (ctrl.event)
        {
```

```
            :
        }
    }
}
```

4.11 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 8, Configuration (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. If the value other than *USB_ON* or *USB_OFF* is assigned, then *USB_ERR_PARA* will be the return value. When *USB_NULL* is set to the 2nd argument *(state)*, this driver operates the same processing as when *USB_OFF* is set.

5. If *USB_NULL* is assigned to the argument *(p_ctrl)*, then *USB_ERR_PARA* will be the return value.

6. Use this API only when the application program needs to control VBUS. (This driver does not control VBUS after turning on VBUS in the initialization processing.)

7. 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.

8. Do not call this API in the following function.

   (1).   Interrupt function.

   (2).   Callback function regsitered by *R_USB_Callback* function.

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 */
            :
            :

}
```

4.12   R_USB_PullUp

Pull-up enable/disable setting of D+/D- line

Format

usb_err_t   R_USB_PullUp(uint16_t state)

Arguments

state   Pull-up enable/disable setting

Return Value

USB_SUCCESS   Successful completion (Pull-up enable/disable setting completed)
USB_ERR_PARA   Parameter error
USB_ERR_NG   Other error

Description

This API enables or disables pull-up of D+/D- line.

Note

1. Assign "*USB_ON*" or "*USB_OFF*" to the argument(*state*). Assign "*USB_ON*" in order to enable pull-up, and assign "*USB_OFF*" in order to disable pull-up. If the value other than *USB_ON* or *USB_OFF* is assigned, then *USB_ERR_PARA* will be the return value. When *USB_NULL* is set to the argument, this driver operates the same processing as when *USB_OFF* is set.

2. Use this API only when need the control of D+/D- line in the application program. (USB driver controls D+/D- line when attaching or detaching to USB Host)

3. This API is processed only in USB Peripheral mode. If this API is called in USB Host mode, then *USB_ERR_NG* will be the return value.

4. Do not call this API in the following function.

   (1).   Interrupt function.

   (2).   Callback function regsitered by *R_USB_Callback* function.

Example

```
void usb_peri_application( void )
{
            :
            :
    R_USB_PullUp(USB_ON ); /* Pull-up enable */
            :
            :
    R_USB_PullUp(USB_OFF ); /* Pull-up disable */
            :
            :
}
```

4.13 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
    USB_ERR_PARA   Parameter error
    USB_ERR_NG   Other error

Description

This function gets the USB device information.

For information to be gotten, see chpater 9.6, usb_info_t structure.

Note

1. Call this API after calling the *R_USB_Open* function (and before calling the *R_USB_Close* function). When this API is called before calling *R_USB_Open* function, then *USB_ERR_PARA* will be the return value.

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*).

5. In USB Host mode, do not assign *USB_NULL* to the first arugument (*p_ctrl*). If *USB_NULL* is assigned, then *USB_ERR_PARA* will be the return value.

6. Do not assign *USB_NULL* to the second arugument (*p_info*). If *USB_NULL* is assigned, then *USB_ERR_PARA* will be the return value.

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 );
            :
}
```

4.14 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

1. Non-OS

   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.

2. RTOS

   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*).
   It is possible to check for the completion of a data read based on an argument (*USB_STS_READ_COMPLETE* in the member (*event*) of the *usb_ctrl_t* structure) to the callback function registered in the USB driver.
   After confirming the argument to the callback function registered in the USB driver (*USB_STS_READ_COMPLETE* in the event member of the *usb_ctrl_t* structure), reference the size member of the *usb_ctrl_t* structure to ascertain the size of the data from the completed read.

Note

1. Please specify a multiple of MaxPacketSize to the 3rd argument (*size*).

2. This API only performs data read request processing. An application program does not wait for data read completion by using this API.

3. 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.

4. 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. (Non-OS)

5. If the data that has been read is not n times the maximum packet size and does not satisfy the read request size, the USB driver will assume that the data transfer is still in progress, so it will not call the callback function that provides notification that data reception is complete. (RTOS)

6. 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.

7. 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.

8. 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.

9. Do not assign a pointer to the auto variable (stack) area to the second argument (*p_buf*).

10. Specify the following address for the 2nd argument (*p_buf*) when using DMA/DTC transfer.

   (1). When using USBA/USBAa module

   Start address of the buffer area aligned on 4-byte boundary.

   (2). When using other than USBA/USBa module

   Start address of the buffer area aligned on 2-byte boundary.

11. 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)*. Allocate the area of the following size when using DTC/DMA transer.

   (1). When *USB_CFG_CNTMDON* is specified for *USB_CFG_CNTMD* definition in *r_usb_basic_config.h* (When using USBA/USBAa module.)

   Allocate the area more than n times FIFO buffer size. For FIFO buffer size, refer to the chapter 12.4, Change of PIPEBUF Register

   (2). When *USB_CFG_CNTMDOFF* is specified for *USB_CFG_CNTMD* definition in *r_usb_basic_config.h*.

   Allocate the area n times the max packet size.

12. If 0 (zero) is assigned to one of the arguments, then *USB_ERR_PARA* will be the return value.

13. 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. (Non-OS)

14. 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.

15. 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.

16. To transfer the data for a Control transfer, use the *R_USB_Read* function rather than this API.

17. Enable one of *USB_CFG_HVND_USB* or *USB_CFG_PVND_USE* definition when using this API. If this API is used when these definitions are not enabled, *USB_ERR_NG* is returned. For *USB_CFG_HVND_USB* or *USB_CFG_PVND_USE* definition, refer to chapter 8, Configuration.

18. 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.

19. Do not call this API in the following function.

   (1). Interrupt function.

   (2). Callback function regsitered by *R_USB_Callback* function.

Example

1. Non-OS

```
uint8_t      g_buf[512];
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, g_buf, size);
                    :
            break;
            case USB_STS_READ_COMPLETE:
                    :
            break;
                    :
        }
    }
}
```

2. RTOS

```c
uint8_t          g_buf[512];
/* Callback Function */
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
     USB_APL_SND_MSG(task_id, (usb_msg_t *)p_ctrl);
}

/* Application Task */
void usb_application_task( void )
{
    usb_ctrl_t ctrl;
    usb_ctrl_t *p_mess;
               :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
        {
                    :
            case USB_STS_WRITE_COMPLETE:
                    :
                ctrl.module = USB_IP1;
                ctrl.pipe = USB_PIPE1;
                R_USB_PipeRead(&ctrl, g_buf, size);
                    :
            break;
            case USB_STS_READ_COMPLETE:
                    :
            break;
                    :
        }
    }
}
```

4.15   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

1.   Non-OS

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*).

2.   RTOS

This function requests a data write (bulk/interrupt transfer).
The write data is stored in the area specified in the argument (*p_buf*).
It is possible to check for the completion of a data write based on an argument (*USB_STS_WRITE_COMPLETE* in the member (*event*) of the *usb_ctrl_t* structure) to the callback function registered in the USB driver.
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.

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.   Specify the following address for the 2nd argument (*p_buf*) when using DMA/DTC transfer.

(1).   When using USBA/USBAa module

Start address of the buffer area aligned on 4-byte boundary.

(2).   When using other than USBA/USBa module

Start address of the buffer area aligned on 2-byte boundary.

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. (Non-OS)

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. Enable one of *USB_CFG_HVND_USB* or *USB_CFG_PVND_USE* definition when using this API. If this API is used when these definitions are not enabled, *USB_ERR_NG* is returned. For *USB_CFG_HVND_USB* or *USB_CFG_PVND_USE* definition, refer to chapter 8, Configuration.

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.

15. Do not call this API in the following function.

  (1). Interrupt function.

  (2). Callback function regsitered by *R_USB_Callback* function.

Example

1.   Non-OS

```
uint8_t    g_buf[512];
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, 512);
                        :
            break;
            case USB_STS_WRITE_COMPLETE:
                        :
            break;
                        :
        }
    }
}
```

2. RTOS

```
uint8_t       g_buf[512];
/* Callback Function */
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
      USB_APL_SND_MSG(task_id, (usb_msg_t *)p_ctrl);
}

/* Application Task */
void usb_application_task( void )
{
    usb_ctrl_t ctrl;
    usb_ctrl_t *p_mess;
                 :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
        {
                         :
            case USB_STS_READ_COMPLETE:
                         :
                ctrl.moudle = USB_IP0;
                ctrl.pipe = USB_PIPE2;
                R_USB_PipeWrite(&ctrl, g_buf, 512);
                         :
            break;
            case USB_STS_WRITE_COMPLETE:
                         :
            break;
                         :
        }
    }
}
```

4.16   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_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. If *USB_NULL* is assigned to the argument (*p_ctrl*), then *USB_ERR_PARA* will be the return value.

5. In USB Host mode, *USB_ERR_NG* will be the return value when this API can not stop the data read/write request.

6. When the *R_USB_GetEvent* function is called after a data read/write stopping has been completed, the return value *USB_STS_READ_COMPLETE/USB_STS_WRITE_COMPLETE* is returned. (Non-OS)

7. USB driver set *USB_STS_READ_COMPLETE* or *USB_STS_WRITE_COMPLTE* to the argument (the member (*event*) of the *usb_ctrl_t* structure) in the callback function registered in the USB driver when a data read/write stopping has been completed. (RTOS)

8. 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.

9. Enable one of *USB_CFG_HVND_USB* or *USB_CFG_PVND_USE* definition when using this API. If this API is used when these definitions are not enabled, *USB_ERR_NG* is returned. For *USB_CFG_HVND_USB* or *USB_CFG_PVND_USE* definition, refer to chapter 8, Configuration.

10. 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.

9. Do not call this API in the following function.

   (1).   Interrupt function.

   (2).   Callback function regsitered by *R_USB_Callback* function.

Example

1.　Non-OS

```
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;
                        :
        }
    }
}
```

2.　RTOS

```
/* Callback Function */
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
    USB_APL_SND_MSG(task_id, (usb_msg_t *)p_ctrl);
}
/* Application Task */
void        usb_application_task( void )
{
    usb_ctrl_t ctrl;
    usb_ctrl_t *p_mess;
                :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
        {
                        :
            case USB_STS_DETACH:
                        :
                ctrl.module = USB_IP0;
                ctrl.pipe = USB_PIPE1;
                R_USB_PipeStop( &ctrl );
                        :
                break;
                        :
        }
    }
}
```

4.17　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*). In USB Host mode, does not assigne *USB_NULL* to the first argument (*p_ctrl*). f *USB_NULL* is assigned, then *USB_ERR_PARA* will be the return value.

4.　Bit map information b0(PIPE0) is always set to "1".

5.　If *USB_NULL* is assigned to the argument (*p_ctrl*), 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

(1). In the case of USB Host mode

```
void     usb_application( void )
{
    uint16_t usepipe;
    usb_ctrl_t ctrl;
                        :
    while (1)
    {
                    :
            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)
    {
                    :
            case USB_STS_CONFIGURED:
                    :
                R_USB_GetUsePipe((usb_ctrl_t *)USB_NULL, &usepipe);
                    :
            break;
                    :
    }
}
```

4.18   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. If 0 (zero) is assigned to one of the arguments, *USB_ERR_PARA* will be the return value.

5. In USB Peripheral mode, no assignment to the members (*address* and *module*) is required.

6. Refer to chapter 9.5, usb_pipe_t structure for details on the *usb_pipe_t* structure.

7. 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)
    {
                      :
            case USB_STS_CONFIGURED:
                      :
                ctrl.pipe = USB_PIPE3;
                ctrl.module = USB_IP1;
                ctrl.address= address;
                R_USB_GetPipeInfo( &ctrl, &info );
                      :
            break;
                      :
    }
}
```

5.   Callback Function (FreeRTOS, uITRON only)

When a USB event has completed, the USB driver will call a callback function. A callback function is created by the user as a user application program, and must be registered in the USB driver using the *R_USB_Callback* function.

A callback functions that is registered in the USB driver must support the arguments and return values shown below.

```
Arguments     :  usb_ctrl_t      *p_ctrl       // Pointer to a usb_ctrl_t structure area
              :  rtos_task_id_t  task_id       // Task handle which USB event has completed
              :  uint8_t         is_request    // Class request reception flag
Return values :  void                          // None
```

Note:

(1).   In addition to the USB completion event, a variety of information about the event is also set to the argument (*p_ctrl*) by the USB driver. Be sure to notify the application task of the relevant argument information using the real-time OS API.

(2).   If the member (*event*) in the argument (*p_ctrl*) is the following, the task ID of the application task which calls the API related to the event is set to the argument (*task_id*). In the other case, *USB_NULL* is set to the argument (*task_id*).

      a.   USB_STS_READ_COMPLETE

      b.   USB_STS_WRITE_COMPLETE

      c.   USB_STS_REQUEST_COMPLETE (Note a)

      d.   USB_STS_SUSPEND (Note b)

      e.   USB_STS_RESUME (Note b)

      f.   USB_STS_MSC_CMD_COMPLETE

   Note:

      a.   In USB Peripheral mode, when this driver received the class request with the no data status stage, *USB_NULL* is set to the argument(*task_id*).

      b.   In USB Peripheral mode, *USB_NULL* is set to the argument(*task_id*).

(3).   In USB Peripheral mode, when this driver received the class request, *USB_ON* is set to the argument (*is_request*).In the other case, *USB_OFF* is set.When the argument (*is_request*) is *USB_ON*, the information related the class request is set to the member (*setup*) in the argumet (*p_ctrl*).

Example)

```
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
    /* Notify application task of USB event information using the real-time OS API */
    USB_APL_SND_MSG(USB_APL_MBX, (usb_msg_t *)p_ctrl);
}
```

6.   Return Value of R_USB_GetEvent Function / Retrieval of USB Completion Events (Except for Azure RTOS)

(1). Non-OS

The return values for the *R_USB_GetEvent* function are listed in **Table 6-1, Return Value of R_USB_GetEvent Function / Retrieval of USB Completion Events**. Make sure you describe a program in the application program to be triggered by each return value from the *R_USB_GetEvent* function.

(2). FreeRTOS, uITRON

When a USB event has completed, the callback function that has been registered using the *R_USB_Callback* function will be called by the USB driver. The member (*event*) of the argument to this callback function (the pointer to the *usb_ctrl_t* structure) will be set to the USB event information for the completed event. In the application program, be sure to define a callback function and, from inside that callback function, notify the application task of the completed USB event by using API etc supported by the real time OS.

Table 6-1 Return Value of R_USB_GetEvent Function / Retrieval of USB Completion Events

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 (Non-OS only)	No USB-related events.	○	○

6.1   USB_STS_DEFAULT

Indicates that the device state of the USB device has transitioned to the Default state.

6.2   USB_STS_CONFIGURED

Indicates that the device state of the USB device has transitioned to the Configured state. The *usb_ctrl_t* structure is set t the following information. In USB host mode, information is also set in the following *usb_ctrl_t* structure member.

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.

6.3   USB_STS_SUSPEND

Indicates that the device state of the USB device has transitioned to the Suspend state.

6.4   USB_STS_RESUME

Indicates that the USB device in the Suspend state has been resumed from the Suspend state by the Resume signal.

Note:

When in USB Host mode, indicates that the USB device has been resumed by the RemoteWakeUp signal from an HID device.

6.5   USB_STS_DETACH

Indicates that the USB device is in the Detached state from USB Host. When in USB Host mode, information is also set to the following members of the *usb_ctrl_t* structure.

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)

6.6   USB_STS_REQUEST

Indicates the state in which the USB device has received a USB request (Setup). Information is also set to the following members of the *usb_ctrl_t* structure.

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*. (Non-OS)

2.   If a request for support of the no data control status stage is received, the *event* member will be set to *USB_STS_REQUEST_COMPLETE*, and not to *USB_STS_REQUEST*. (RTOS)

3.   For more details on USB request information (8 bytes) stored in member (*setup)*, refer to chapter 9.2, usb_setup_t structure.

6.7   USB_STS_REQUEST_COMPLETE

Indicates that the stage transits to the idle stage after the status stage of a control transfer is completed. 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 9.2, usb_setup_t structure.

6.8   USB_STS_READ_COMPLETE

Indicates that a data read has been completed by *R_USB_Read* / *R_USB_PipeRead*. 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 7, 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.

(3). The read data size is set in the member *size* when the read completion error information is *USB_SUCCESS* or *USB_ERR_SHORT*.

6.9 USB_STS_WRITE_COMPLETE

Indicates that a data read has been completed by *R_USB_Write* / *R_USB_PipeWrite*. 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 7, 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

6.10 USB_STS_BC

Indicates the state in which a USB Host / USB device that supports the battery charging feature has been connected. 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)

6.11 USB_STS_OVERCURRENT

In USB Host mode, indicates that the overcurrent is detected. Information is also set in the following *usb_ctrl_t* structure member.

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

6.12 USB_STS_NOT_SUPPORT

In USB Host mode, indicates that an unsupported USB device is connected, then *USB_STS_NOT_SUPPORT* will be the return value.

6.13 USB_STS_NONE (Non-OS)

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

7. Device Class Types (Except for Azure RTOS)

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_HVND	Host Vendor Class
USB_PVND	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.

8.   Configuration (r_usb_basic_config.h)

8.1   USB Host and Peripheral Common Configurations

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*.

(1).   USB Host mode

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

#define   USB_CFG_MODE   USB_CFG_HOST

(2).   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_HVND_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_PVND_USE	// Peripheral Vendor Class
#define	USB_CFG_PCDC_2COM_USE	// Peripheral Composite device(CDC VCOM 2Port)
#define	USB_CFG_PCDC_PHID_USE	// Peripheral Composite device(CDC + HID)
#define	USB_CFG_PCDC_PMSC_USE	// Peripheral Composite device(CDC + MSC)
#define	USB_CFG_PHID_PMSC_USE	// Peripheral Composite device(HID + MSC)

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:

(1).   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.

(2).   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:

(1).   Set one of the DMA channel numbers from *USB_CFG_CH0* to *USB_CFG_CH7*. Do not set the same DMA Channel number.

(2).   If DMA transfer is not used, set *USB_CFG_NOUSE* as the DMA Channel number.

(3).   Be sure to specify the different DMA channel number to DMA sending and receiving when usng USB Host Mass Storage classs.

The following is the spcifying example.

a.   When using the DMA transfer for DMA sending and receiving by using USB0 module

    #define   USB_CFG_USB0_DMA_TX   USB_CFG_CH0
    #define   USB_CFG_USB0_DMA_RX   USB_CFG_CH3

    Note:

    Be sure to specify USB PIPE1 and USB PIPE2 for DMA transfer.

b.   When using DMA for data sending and not using DMA for data receiving using USB1 module

    #define   USB_CFG_USB1_DMA_TX   USB_CFG_CH0

    Note:

    Specify the one of USB PIPE1 or USB PIPE2 for the sending USB PIPE (DMA transfer) and specify the one of USB_PIPE3, USB_PIPE4 or USB_PIPE5 for the receiving USB PIPE.

c.   When using DMA for data sending and not using DMA for data receiving using USB0 module, and, when using DMA for data receiving and not using DMA for data sending using USB1 module.

    #define   USB_CFG_USB0_DMA_TX   USB_CFG_CH1
    #define   USB_CFG_USB1_DMA_RX   USB_CFG_CH2

    Note:

    a)   In USB0 module, specify the one of USB PIPE1 or USB PIPE2 for the sending USB PIPE (DMA transfer) and specify the one of USB_PIPE3, USB_PIPE4 or USB_PIPE5 for the receiving USB PIPE.

    b)   In USB1 module, specify the one of USB PIPE1 or USB PIPE2 for the receiving USB PIPE (DMA transfer) and specify the one of USB_PIPE3, USB_PIPE4 or USB_PIPE5 for the sending USB PIPE.

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. 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.

9. 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:

(1). 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.

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

(3). The USBAa or USBA module is a USB module used in the RX71M or RX64M.

10. Interrupt Priority Level setting

Assign the interrupt priority level of the interrupt related to USB for *USB_CFG_INTERRUPT_PRIORITY* definition.

#define	USB_CFG_INTERRUPT_PRIORITY	3	// 1(low) – 15(high)

8.2   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. USB Hub use setting

Specify whether to use the USB Hub.

#define	USB_CFG_HUB	USB_CFG_ENABLE	// Using USB Hub
#define	USB_CFG_HUB	USB_CFG_DISABLE	// Not using USB Hub

4.

5. Type-C use setting

   Specify whether to use the Type-C.

#define	USB_CFG_TYPEC	USB_CFG_ENABLE	// Using Type-C
#define	USB_CFG_TYPEC	USB_CFG_DISABLE	// Not using Type-C

6. 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.

7. 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.

8. 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.

9. 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:

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

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

8.3 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. Request notification setting

Set whether this driver notifies the application program the reception of *SET_INTERFACE,*
SET_FEATURE/CLEAR_FEATURE request or not.

#define	USB_CFG_REQUEST	USB_CFG_ENABLE	// Notification
#define	USB_CFG_REQUEST	USB_CFG_DISABLE	// Not notification

Note:

This driver notifies the application program the request reception when Feature Selector (wValue) in the received *SET_FEATURE/CLEAR_FEATURE* is *DEVICE_REMOTE_WAKEUP*.

8.4   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 PIPE1 to PIPE5 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).   Please set *USB_CFG_CNTMDOFF* to the *USB_CFG_CNTMD* definition when supporting Host Mass Storage Class and Peripheral Mass Storage Class.

9. Structures

This chapter describes the structures used in the application program.

Note:

Don't use the structure other than *usb_cfg_t* structure when using Azure RTOS.

9.1 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*.

```
typedef struct usb_ctrl {
    uint8_t        module;        /* Note 1 */
    uint8_t        address;       /* Note 2 */
    uint8_t        pipe;          /* Note 3 */
    uint8_t        type;          /* Note 4 */
    uint8_t        status;        /* Note 5 */
    uint8_t        event          /* Note 6 */
    uint32_t       size;          /* Note 7 */
    usb_set_up     setup;         /* Note 8 */
    void           *p_data;       /* Note 9 */
} 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 status stage processing to a vendor class request, see chapter 10.2.3, Status Stage Processing.

6. Member (*event*) is used to set the completion event by USB driver. (RTOS only)

7. 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.

8. Member (*setup*) is used to set the information about a class request.

9. Member (*p_data*) is used to set information other than the above.

9.2 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.

9.3   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 9.4, 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.

9.4   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    **pp_string;    /* Note 5 */
    uint8_t    num_string;     /* Note 6 */
} 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 (*pp_string)*. In the string descriptor table, specify the top address of the areas that store each string descriptor.

<table>
<tr><td>Ex. 1) Full-speed</td><td>Ex. 2) Hi-speed</td></tr>
<tr><td>usb_descriptor_t usb_descriptor =</td><td>usb_descriptor_t usb_descriptor =</td></tr>
<tr><td>{</td><td>{</td></tr>
<tr><td>   smp_device,</td><td>   smp_device,</td></tr>
<tr><td>   smp_config_f,</td><td>   smp_config_f,</td></tr>
<tr><td>   USB_NULL,</td><td>   smp_config_h,</td></tr>
<tr><td>   USB_NULL,</td><td>   smp_qualifier,</td></tr>
<tr><td>   smp_string,</td><td>   smp_string,</td></tr>
<tr><td>   3,</td><td>   3,</td></tr>
<tr><td>};</td><td>};</td></tr>
</table>

6. Specify the number of the string descriptor which set in the string descriptor table to the member (*num_string)*.

9.5 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.

```
typedef 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)*.

9.6 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)*. In USB Peripheral mode, the supporting device class type is set for the member (*type)*. For information on the device class types, see 7, 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)
```

9.7　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.

10.　　USB Class Requests (Except for Azure RTOS)

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.

10.1　　USB Host operations

10.1.1　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 9.2, 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.

10.1.2　USB request completion

1. Non-OS

   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.

2. FreeRTOS, uITRON

   It is possible to check for the completion of a USB request based on an argument to the callback function registered in the USB driver (*USB_STS_REQUEST_COMPLETE* in the event member of the *usb_ctrl_t* structure). In the case of a request for support of the control read data stage, the received data will be stored in the area specified by the second *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

10.1.3　USB request processing example

1. Non-OS

```
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;
              }
      }

  2.   FreeRTOS, uITRON

  void usr_application_task (void )
  {
      usb_ctrl_t ctrl;
      usb_ctrl_t *p_mess;
                  :
      while(1)
      {
          USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
          ctrl = *p_mess;
          switch(ctrl.event)
          {
              /* 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;
          }
      }
  }
```

10.2   USB Peripheral operations

10.2.1 USB request (Setup)

1.   Non-OS

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 9.2, usb_setup_t structure for a description of the settings for member (*setup*).

2.   FreeRTOS, uITRON

It is possible to check for the reception of a USB request (Setup) transmitted from the USB Host using an argument to the callback function registered in the USB driver (*USB_STS_REQUEST* in the event member of the *usb_ctrl_t* structure). The contents of the USB request (Setup: 8 bytes) are stored in the *usb_ctrl_t* structure member (*setup*) area. Refer to chapter 9.2, usb_setup_t structure for a description of the settings for member (*setup*).

Note:

Note that when a request for support of the no data status stage is received, the argument to the callback function registered in the USB driver (the member (*event*) of the *usb_ctrl_t* structure) will be set to *USB_STS_REQUEST_COMPLETE*, and not to *USB_STS_REQUEST*.

10.2.2 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:

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

  (2). It is possible to check for the reception completion of a request data based on an argument to the callback function registered in the USB driver (*USB_STS_READ_COMPLETE* in the member (*event*) of the *usb_ctrl_t* structure). (RTOS)

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:

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

  (2). It is possible to check for the transmission completion of a request data based on an argument to the callback function registered in the USB driver (*USB_STS_REQUEST_COMPLETE* in the member (*event*) of the *usb_ctrl_t* structure). (RTOS)

10.2.3 Status Stage Processing

In the following case, this driver does not process to the status stage. The user need to process the status stage in the application program. The user needs to process the status stage in the application program. For how to process the status stage, see 10.2.4, Example USB request processing description.

  (1). Case of responding ACK to a class request that supports no-data control status stage

  (2). Case of responding STALL to a class request

Note:

When receiving a class r equest that support the data stage, this USB driver process the status stage after processing the data stage.

10.2.4 Example USB request processing description

1. Request that supports control read data stage

    (1). Non-OS

```
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:
                        :
        break;
                        :

    }
}
```

    (2). FreeRTOS, uITRON

```
void usr_application_task (void )
{
    usb_ctrl_t ctrl;
    usb_ctrl_t *p_mess;
            :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
    {
                        :
        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:
                        :
        break;
                        :
        }
    }
}
```

2.   Request that supports control write data stage

   (1). Non-OS

```
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;
    }
}
```

   (2). FreeRTOS, uITRON

```
void usr_application_task (void )
{
    usb_ctrl_t ctrl;
    usb_ctrl_t *p_mess;
                :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
        {
                            :
            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

   (1). ACK response

      a. Non-OS

```
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:
                                    :
            break;
                                    :
        }
    }
```

b. FreeRTOS, uITRON

```
void usr_application_task (void )
{
    usb_ctrl_t *p_mess;
                :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
        {
                                    :
            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:
                                    :
            break;
                                    :
        }
    }
}
```

(2). STALL response

   a. Non-OS

```
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:
                                    :
        break;
                                    :
    }
}
```

b. FreeRTOS, uITRON

```
void usr_application_task (void )
{
    usb_ctrl_t *p_mess;
            :
    while(1)
    {
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t **)&p_mess);
        ctrl = *p_mess;
        switch (ctrl.event)
        {
                :
            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:
                    :
            break;
                    :
        }
    }
}
```

11.   DTC/DMA Transfer

11.1   Basic Specification

The specifications of the DTC/DMA transfer sample program code included in USB-BASIC-FW are listed below.
USB Pipe 1 and Pipe2 can used DTC/DMA access.
Table11-1 shows DTC/DMA Setting Specifications.

Table11-1   DTC/DMA Setting Specifications

Setting	Description
FIFO port used	D0FIFO and D1FIFO port
Transfer mode	Block transfer mode
Chain transfer	Disabled
Address mode	Full address mode
Read skip	Disabled
Access bit width (MBW)	4-byte transfer: 32-bit width (when using USB1 module only)
	2-byte transfer: 16-bit width (when using USB0 module only)
USB transfer type	BULK transfer
Transfer end	Receive direction: BRDY interrupt
	Transmit direction: D0FIFO/D1FIFO interrupt, BEMP interrupt

Note:

   This driver does not support to use DMA transfer and DTC transfer at the same time.

11.2   Notes

11.2.1   DTC transfer

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

2.   Please add the following 3 lines immediately before including the *r_dtc_rx_if.h* file in *r_usb_basic¥src¥driver¥ r_usb_usbif_api.c* file and *r_usb_basic¥src¥hw¥r_usb_dma.c* file when using DTC transfer of RX66T.

```
#if    defined(BSP_MCU_RX66T)        // Addition
#undef     BSP_MCU_RX72T            // Addition
#endif                              // Addition
#include    "r_dtc_rx_if.h"
```

11.2.2   Data Reception Buffer Size

The user needs to allocate the buffer area for the following size to store the receiving data.

   (1).   When *USB_CFG_CNTMDON* is specified for *USB_CFG_CNTMD* definition in *r_usb_basic_config.h* (When using USBA/USBAa module.)

      Allocate the area more than n times FIFO buffer size. For FIFO buffer size, refer to the chapter 12.4, Change of PIPEBUF Register

   (2).   When *USB_CFG_CNTMDOFF* is specified for *USB_CFG_CNTMD* definition in *r_usb_basic_config.h*.

      Allocate the area n times the max packet size.

11.2.3   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 for DMA/DTC transfer. When data transfer is performed by combining DMA/DTC transfer and CPU transfer, use PIPE1 or PIPE2 for DTM/DTC transfer and use PIPE3, PIPE4 or PIPE5 for CPU transfer.

12. Additional Notes

12.1 Vendor ID

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

12.2 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 9.7, 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 14.1, usb_compliance_disp function.

12.3 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 chapter 8, Configuration (r_usb_basic_config.h).

12.4 Change of PIPEBUF Register

Recommended values are set to the BUFSIZE and BUFNMB bits of the PIPEBUF register that are supported by the USBA and USBAa modules. When changing to these bits, change the *usb_cstd_get_pipe_buf_value* function *(r_usb_cdataio.c* file) in the USB driver.

12.5 RTOS

12.5.1 FreeRTOS

1. Task Priority

The value from 8 to 11 is specified to the priority for USB driver task. Please specify the value from 0 to 7 to the priority for the application task.

2. configMAX_PRIORITIES definition

Specify a value of 12 or more to *configMAX_PRIORITIES* definition defined in *FreeRTOSConfig.h* file when using FreeRTOS.

12.5.2 RI600V4 (Configuration File Creation)

When using RI600V4, you need to create a configuration file to register the OS resources used by USB driver with RI600V4. Please create a configuration file based on the following information. For how to create a configuration file, refer to RI600V4 user's manual.

1. USB Peripheral Definition

    (1). Task Definition

name	:	ID_ USB_RTOS_PCD_TSK
entry_address	:	usb_pstd_pcd_task()
stack_size	:	512
initial_start	:	OFF
exinf	:	0

    Note:

    For this task priority, specify a prority higher than the application task priority.

    (2). Mailbox Definition

    a. Mailbox 1

name	:	ID_USB_RTOS_PCD_MBX
wait_queue	:	TA_FIFO
message_queue	:	TA_MFIFO

    b. Mailbox 2

name	:	ID_USB_RTOS_PCD_SUB_MBX
wait_queue	:	TA_FIFO
message_queue	:	TA_MFIFO

    (3). Fixed-size Memory Pool Definition

name	:	ID_USB_RTOS_DRIVER_MPF
section	:	BRI_HEAP
num_block	:	64
siz_block	:	64
wait_queue	:	TA_TFIFO

2. USB Host Definition

    (1). Task Definition

    a. Task 1

name	:	ID_USB_RTOS_HCD_TSK
entry_address	:	usb_hstd_hcd_task()
stack_size	:	512
initial_start	:	OFF
exinf	:	0

    b. Task 2

name	:	ID_USB_RTOS_MGR_TSK
entry_address	:	usb_hstd_mgr_task()
stack_size	:	512
initial_start	:	OFF
exinf	:	0

c. Task 3

name	:	ID_ USB_RTOS_HUB_TSK
entry_address	:	usb_hstd_hub_task()
stack_size	:	512
initial_start	:	OFF
exinf	:	0

Note:

Be sure to specify the task priority in the following order.

ID_USB_RTOS_HCD_TSK	High Priority
ID_USB_RTOS_MGR_TSK	↓
ID_USB_RTOS_HUB_TSK	↓
Application Task	Low Priority

(2). Mailbox Definition

a. Mailbox 1

name	:	ID_USB_RTOS_HCD_MBX
wait_queue	:	TA_FIFO
message_queue	:	TA_MFIFO

b. Mailbox 2

name	:	ID_USB_RTOS_HCD_SUB_MBX
wait_queue	:	TA_FIFO
message_queue	:	TA_MFIFO

c. Mailbox 3

name	:	ID_USB_RTOS_HCD_SUB_ADDR_MBX
wait_queue	:	TA_FIFO
message_queue	:	TA_MFIFO

d. Mailbox 4

name	:	ID_USB_RTOS_MRG_MBX
wait_queue	:	TA_FIFO
message_queue	:	TA_MFIFO

e. Mailbox 5

name	:	ID_USB_RTOS_HUB_MBX
wait_queue	:	TA_FIFO
message_queue	:	TA_MFIFO

f. Mailbox 6

name	:	ID_USB_RTOS_CLS_MBX
wait_queue	:	TA_FIFO
message_queue	:	TA_MFIFO

Note:

It is not necessary to define the mailbox for *ID_USB_RTOS_HUB_MBX* when *USB_CFG_DISABLE* is specified for *USB_CFG_HUB* definition in the *r_usb_basic_config.h* file.

(3). Fixed-size Memory Pool Definition

name	:	ID_USB_RTOS_DRIVER_MPF
section	:	BRI_HEAP
num_block	:	64
siz_block	:	64

wait_queue   :   TA_TFIFO

3. USB Host and USB Peripheral Common Definition

(1). System Definition

stack_size   :   System stack size
priority   :   Maximum task priority
system_IPL   :   Kernel interrupt mask level
tic_deno   :   1
tic_nume   :   1
context   :   FPSW, ACC

(2). System Clock Definition

timer   :   Hardware timer
template   :   Template file
timer_clock   :   PCLK frequency
IPL   :   1 to system.system_IPL

(3). Fixed-size Memory Pool Definition

name   :   ID_USB_RTOS_DRIVER_MPF
section   :   BRI_HEAP
num_block   :   64
siz_block   :   64
wait_queue   :   TA_TFIFO

(4). Interrupt Definition

a. Interrupt 1

entry_address   :   usbfs_usbi_isr() , usbhs_usbar_isr()
os_int   :   YES

Note:

(a). Specify USBI interrupt function (*usbfs_usbi_isr*) for the *entry_address* item when using USB Full-speed module.

(b). Specify USBAR interrupt function (*usbhs_usbar_isr*) for the *entry_address* item when using USB Hi-speed module.

b. Interrupt 2 (When using DMA tranfer)

entry_address   :   r_dmaca_intdmac0i_isr()
     r_dmaca_intdmac1i_isr()
     r_dmaca_intdmac2i_isr()
     r_dmaca_intdmac3i_isr()
     r_dmaca_intdmac74i_isr()
os_int   :   YES

Note:

Specify the following interrupt function in the entry_address item according to the DMA channel number to be used. For example, if *USB_CFG_CH1* is specified for *USB_CFG_USB0_DMA_TX* definition in the *r_usb_basic_config.h file*, specify *r_dmaca_intdmac1i_isr()* in the entry_address item.

DMA Channel Number	Function
DMA0	r_dmaca_intdmac0i_isr()
DMA1	r_dmaca_intdmac1i_isr()
DMA2	r_dmaca_intdmac2i_isr()
DMA3	r_dmaca_intdmac3i_isr()
DMA4 to DMA7	r_dmaca_intdmac74i_isr()

c.   Interrupt 3 (When using DTC transfer)

entry_address	:	usb_cpu_d0fifo_int_hand
		usb_cpu_d1fifo_int_hand
		usb2_cpu_d0fifo_int_hand
		usb2_cpu_d1fifo_int_hand
os_int	:	YES

Note:

For the *entry_address* item, specify the following interrupt function according to the PIPE used.

Use PIPE	Function
PIPE1 (USB0 module)	usb_cpu_d0fifo_int_hand()
PIPE2 (USB0 module)	usb_cpu_d1fifo_int_hand()
PIPE1 (USB1 module)	usb2_cpu_d0fifo_int_hand()
PIPE2 (USB1 module)	usb2_cpu_d1fifo_int_hand()

For example, define the 2 interrupts when using PIPE1 and PIPE2 of USB0 module for the DTC transfer.

Example)

```
interrupt_vector[34] {
    entry_address     =    usb_cpu_d0fifo_int_hand();
    os_int            =    YES;
};


interrupt_vector[35] {
    entry_address     =    usb_cpu_d1fifo_int_hand();
    os_int            =    YES;
};
```

12.5.3　Azure RTOS (USBX)

Please do the following when using Azure RTOS (USBX).

1.　Add componet

Please add the following 3 software components in "Software component configuration" dialog when using Azure RTOS(USBX).

(1).　Azure RTOS USBX XXXX ("XXXX" is device class name).

(2).　r_usb_basic

(3).　r_usb_xxxx ("xxxx" is device class name).

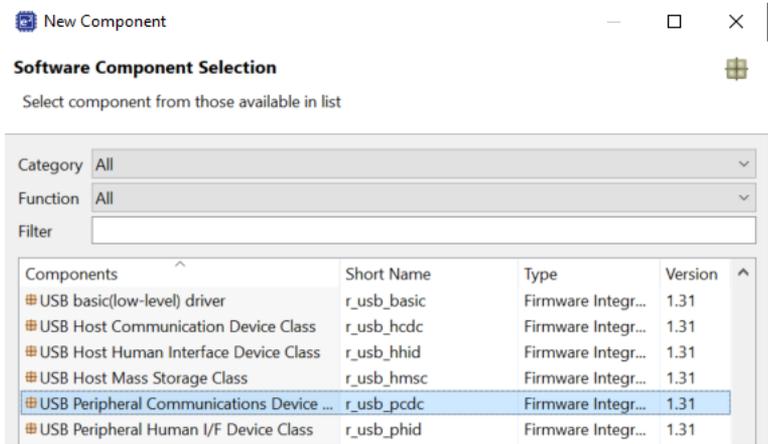The following is the additional example of "Azure RTOS USBX PCDC".

　a.　Selects "Components" tab and push "Add component" button. (**red frame**).
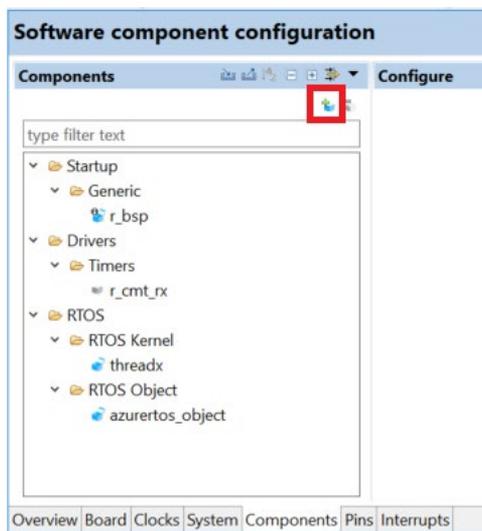
　b.　Selects "Azure RTOS USBX PCDC" and push "Finish" button.
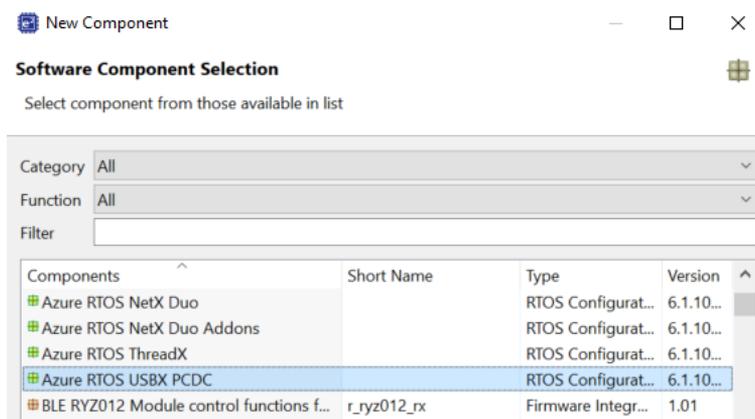
　c.　Selects "USB Peripheral Communication Device Class" and push "Finish" button. (The "r_usb_basic" component is also added.)

d. Confirm that the following 3 componets (**red frame**) are added.
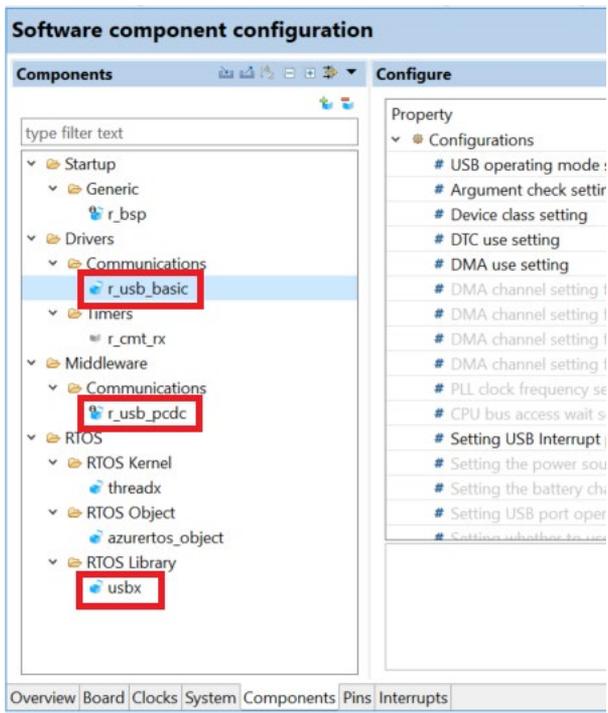
2. **Change Timer tick**

Please change the value of timer tick to "1000" as follow.
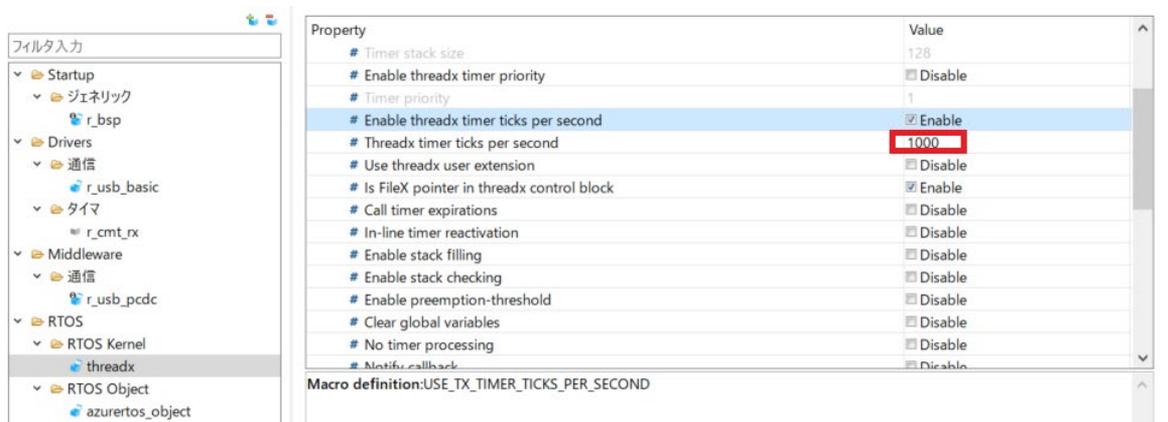
3. **Application Task Priority**

Please specify a priority value for the application task lower than "20".

13. 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.

13.1 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 8, Configuration about setting of the configuration file.

13.2 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.

13.3 Application Program Creation

13.3.1 Include

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

1. r_usb_basic_if.h (Except for Azure RTOS.)

2. r_usb_xxxxx_if.h (Except for Azure RTOS. "xxxx" is device class name.)

3. kernel_id.h (when using RI600V4 only)

4. Include a header file for FAT when creating the application program for Host Mass Storage Class.

5. Include any other driver-related and RTOS related header files that are used within the application program.

> Note:
>
> Please describe "#undef BSP_MCU_RX72T" immediately before including the *r_dtc_rx_if.h* file when using DTC transfer of RX66T.
>
> #undef   BSP_MCU_RX72T
> #include "r_dtc_rx_if.h"

13.3.2 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.

   Table13-1   USB I/O Pin Settings for USB Peripheral Operation

Pin Name	I/O	Function
USB_VBUS	input	VBUS pin for USB communication

   Table13-2   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:
   >
   > Please refer to the corresponding MCU user's manual for the pin settings in ports used for your application program.

2. USB-related initialization

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

3. Creation and Registration of Callback Functions (FreeRTOS, uITRON only)

Use the *R_USB_Callback* function to create callback functions to register. After creation, register the callback function in question in the USB driver using the *R_USB_Callback* function.
In addition to the USB completion event, a variety of information about the event is also set by the USB driver. Be sure to notify the application task of the relevant argument information using the real time OS API etc.

Example)

```
void usb_apl_callback (usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
    /* Notify application task of USB event information using the real time OS API */
    USB_APL_SND_MSG(USB_APL_MBX, (usb_msg_t *)p_ctrl);
}
```

13.3.3　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.

Note:

Please refer to the documentation for Azure RTOS and USBX when using Azure RTOS.

13.3.4　Creation of Main Routine / Application Program Tasks

1. Non-OS

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.

Note:

　a. Carry out USB data communication using the *R_USB_Read*, *R_USB_Write*, *R_USB_PipeRead*, and *R_USB_PipeWrite* functions after checking the return value *USB_STS_CONFIGURED* of *R_USB_GetEvent* function.

　b. Use API supported by FAT when accessing to MSC device in the host mass storage class.

2. FreeRTOS, uITRON

Write application program tasks in loop format. In the main loop, be sure to call the real time OS API to retrieve the information (USB completion events and the like) that is received as notifications from the callback function. Write programs that correspond to the respective USB completion events, with the USB completion events retrieved by the application task as a trigger.

Note:

　a. When the USB device is in the Configured state, carry out USB data communication using the *R_USB_Read*, *R_USB_Write*, *R_USB_PipeRead*, and *R_USB_PipeWrite* functions. It is possible to check whether or not the USB device is in the Configured state using an argument to the callback function registered in the USB driver (*USB_STS_REQUEST_CONFIGURED* in the event member of the *usb_ctrl_t* structure).

　b. Use API supported by FAT when accessing to MSC device in Host mass storage class.

3. Azure RTOS (USBX)

Please refer to the documentation for Azure RTOS and USBX.

13.3.5   Registration to the real time OS (FreeRTOS, uITRON only)

1.   When using the realtime OS other than RI600V4

Register the following in the realtime OS.

(1). Application Program Tasks

(2). The realtime OS features used by application tasks and callback functions

Note:

Set the priority of application program tasks to a priority value of 7 or lower.

2.   When using RI600V4

Create the configuration file. For the configuration file, refer to chapter **12.5.2, RI600V4 (Configuration File Creation).**

13.3.6   Application program description example

(1). Non-OS

```c
#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.p_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;
        }
    }
}
```

(2). FreeRTOS, uITRON

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

/* Callback Function */
void usb_apl_callback(usb_ctrl_t *p_ctrl, rtos_task_id_t task_id, uint8_t is_request)
{
    /* Notify application program task of USB completion event */
    USB_APL_SND_MSG(task_id, (usb_msg_t *)p_ctrl);
}

/* Application Task */
void        usb_application_task(void)
{
    usb_ctrl_t   ctrl;
    usb_ctrl_t   *p_mess;
    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.p_usb_reg = &smp_descriptor; /* Specify the top address of the descriptor table */
    R_USB_Open( &ctrl, &cfg );

    /* Register callback function */
    R_USB_Callback(usb_apl_callback);

    /* main routine */
    while(1)
    {
        /* Retrieve USB completion event about which notification was received from callback
           function */
        USB_APL_RCV_MSG(USB_APL_MBX, (usb_msg_t *)&p_mess);

        ctrl = *p_mess;

        switch(ctrl.event)
        {
            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;
        }
    }
}
```

14. Program Sample

14.1 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

All trademarks and registered trademarks are the property of their respective owners.

Revision Record

Rev.	Date	Page	Summary
			Description
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). 6. Return Value of R_USB_GetEvent Function / Retrieval of USB Completion Events
			(5). 7. Device Class Types
			(6). 8. Configuration
			(7). 9. Structures
			(8). 10. USB Class Requests
			(9). 11. Additional Notes
			(10). 14. Program Sample
			3. The following chapters are deleted.
			"Hub Class", "non-OS Scheduler"
1.22	Sep 30, 2017	—	Supporting RX65N/RX651-2M
1.23	Mar 31,2018	—	1. Supporting Smart Configurator.
			2. Adding the member (num_string) in usb_descriptor_t structure.
1.24	Dec 28, 2018	—	Supported RTOS.
1.25	Apr 16, 2019	—	Added RX66T/RX72T in Target Device.
1.26	May 31, 2019	—	1. Support GCC compiler and IAR compiler.
			2. Remove RX63N from Target Device.
1.27	Jul 31, 2019	—	RX72M is added in Target Device.
1.30	Mar 1, 2020	—	1. Supported the real time OS (uITRON:RI600V4).
			2. Added RX72N/RX66N in Target Device.
1.31	Mar 1, 2021	—	Added RX671 in Target Device.
1.40	Jun 30, 2022	—	Support Azure RTOS (USBX PCDC).
1.41	Oct 30, 2022	—	Support Azure RTOS (USBX HMSC).
1.42	Sep 29, 2023	—	Support Azure RTOS (USBX HCDC).
1.43	Sep 30, 2024	—	Fixed build error on Azure RTOS (USBX PCDC) when using r_usb_basic v.1.42.
1.44	Mar 01, 2025	—	Change Disclaimer.

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. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.

2. Processing at power-on

   The state of the product is undefined at the time 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 time 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 time 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 time when power is supplied until the power reaches the level at which resetting is specified.

3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.

4. 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 the 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.

5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is 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. Additionally, 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.

6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).

7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.

8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.

2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.

3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.

4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.

5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.

6. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

    "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

    "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

    Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.

7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION ("Vulnerability Issues"). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.

8. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.

9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.

10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.

11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.

12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.

13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.

14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:

www.renesas.com/contact/.