RX Family

USB Basic Mini Host and Peripheral Driver (USB Mini Firmware)
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
RX111 Group
RX113 Group
RX231 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. RX111 Group User’s Manual: Hardware (Document number. R01UH0365)
3. RX113 Group User’s Manual: Hardware (Document number. R01UH0448)
4. RX231 Group User’s Manual: Hardware (Document number. R01UH0496)

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

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

1. Overview ........................................................................................................................................... 3
2. Peripheral ......................................................................................................................................... 6
3. Host ................................................................................................................................................ 13
4. API Functions ................................................................................................................................... 19
5. Return Value of R_USB_GetEvent Function ....................................................................................... 48
6. Device Class Types ............................................................................................................................ 51
7. Configuration (r_usb_basic_mini_config.h) ....................................................................................... 52
8. Structures .......................................................................................................................................... 55
9. USB Class Requests ............................................................................................................................ 59
10. DTC/DMA Transfer ............................................................................................................................. 63
11. Additional Notes ................................................................................................................................. 64
12. Creating an Application Program ....................................................................................................... 65
13. Program Sample ................................................................................................................................. 69
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)

<Host mode>
- In host mode, enumeration as Low-speed/Full-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

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.

1.2 Limitations

This driver is subject to the following limitations.
1. In USB host mode, the module does not support suspend during data transfer. Execute suspend only after confirming that data transfer is complete.
2. Multiconfigurations are not supported.
3. Multiinterfaces are not supported.
4. The USB host and USB peripheral modes cannot operate at the same time.
5. USB Hub can not be connected in USB host mode.
6. DMA can not be used when using RX111 and RX113.
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. This driver does not support the CPU transfer using D0FIFO/D1FIFO register.

1.3 Terms and Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APL</td>
<td>Application program</td>
</tr>
<tr>
<td>CDP</td>
<td>Charging Downstream Port</td>
</tr>
<tr>
<td>DCP</td>
<td>Dedicated Charging Port</td>
</tr>
<tr>
<td>HBC</td>
<td>Host Battery Charging control</td>
</tr>
<tr>
<td>HCD</td>
<td>Host control driver of USB-BASIC-F/W</td>
</tr>
<tr>
<td>HDCD</td>
<td>Host device class driver (device driver and USB class driver)</td>
</tr>
<tr>
<td>H/W</td>
<td>Renesas USB device RX Family</td>
</tr>
<tr>
<td>MGR</td>
<td>Peripheral device state manager of HCD</td>
</tr>
<tr>
<td>PBC</td>
<td>Peripheral Battery Charging control</td>
</tr>
<tr>
<td>PCD</td>
<td>Peripheral control driver of USB-BASIC-F/W</td>
</tr>
<tr>
<td>PDCD</td>
<td>Peripheral device class driver (device driver and USB class driver)</td>
</tr>
<tr>
<td>RSK</td>
<td>Renesas Starter Kits</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>USB-BASIC-F/W</td>
<td>USB Basic Mini Host and Peripheral Driver</td>
</tr>
<tr>
<td>Scheduler</td>
<td>Used to schedule functions, like a simplified OS.</td>
</tr>
<tr>
<td>Task</td>
<td>Processing unit</td>
</tr>
</tbody>
</table>
1.4 USB-BASIC-F/W FIT module

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

1.5 Software Configuration

In peripheral mode, USB-BASIC-F/W comprises the peripheral driver (PCD), and the application (APL). PDCD is the class driver and not part of the USB-BASIC-F/W. See Table 1-1. In host mode, USB-BASIC-F/W comprises the host driver (HCD), the manager (MGR) and the application (APL). HDD and HDCD are not part of the USB-BASIC-F/W, see Table 1-1.

The peripheral driver (PCD) and host driver (HCD) initiate hardware control through the hardware access layer according to messages from the various tasks or interrupt handler. They also notify the appropriate task when hardware control ends, of processing results, and of hardware requests.

Manager manages the connection state of USB peripherals and performs enumeration. In addition, manager issues a message to host driver when the application changes the device state.

The customer will need to make a variety of customizations, for example designating classes, issuing vendor-specific requests, making settings with regard to the communication speed or program capacity, or making individual settings that affect the user interface.

![User Programming Layer (UPL)](image)

**Figure 1-1 USB Module Configuration of USB-BASIC-F/W**

<table>
<thead>
<tr>
<th>Table 1-1 Software function overview</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No</strong></td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
1.6 Scheduler Function

A scheduler function manages requests generated by tasks and hardware according to their relative priority. When multiple task requests are generated with the same priority, they are executed using a FIFO configuration. 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 this module before calling `R_USB_Open` function.
2. Peripheral

2.1 Peripheral Control Driver (PCD)

2.1.1 Basic functions

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

PCD accomplishes the following:

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

2.1.2 Issuing requests to PCD

API functions are used when hardware control requests are issued to the PCD and when performing data transfers. Refer to chapter 4, API Functions for the API function.

2.1.3 USB requests

This driver supports the following standard requests.

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

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

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

This Driver API follows the Renesas API naming standards.

2.2.1 Hardware Requirements

This driver requires your MCU support the following features:

- USB

2.2.2 Software Requirements

This driver is dependent upon the following packages:

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

2.2.3 Operating Confirmation Environment

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrated Development</td>
<td>Renesas Electronics e² studio V.7.0.0</td>
</tr>
<tr>
<td>Environment</td>
<td></td>
</tr>
<tr>
<td>C compiler</td>
<td>Renesas Electronics C/C++ compiler for RX Family V.2.08.00</td>
</tr>
<tr>
<td></td>
<td>Compile Option: -lang = c99</td>
</tr>
<tr>
<td>Endian</td>
<td>Little Endian, Big Endian</td>
</tr>
<tr>
<td>USB Driver Revision Number</td>
<td>Rev.1.10</td>
</tr>
<tr>
<td>Using Board</td>
<td>Renesas Starter Kit for RX111</td>
</tr>
<tr>
<td></td>
<td>Renesas Starter Kit for RX113</td>
</tr>
<tr>
<td></td>
<td>Renesas Starter Kit for RX231</td>
</tr>
<tr>
<td>Host Environment</td>
<td>The operation of this USB Driver module connected to the following OSes has been confirmed.</td>
</tr>
<tr>
<td></td>
<td>1. Windows® 7</td>
</tr>
<tr>
<td></td>
<td>2. Windows® 8.1</td>
</tr>
<tr>
<td></td>
<td>3. Windows® 10</td>
</tr>
</tbody>
</table>

2.2.4 Usage of Interrupt Vector

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

<table>
<thead>
<tr>
<th>Device</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX111</td>
<td>USB10 Interrupt (Vector number: 36) / USBR0 Interrupt (Vector number: 90)</td>
</tr>
<tr>
<td>RX113</td>
<td>USB D0FIFO0 Interrupt (Vector number: 36) / USB D1FIFO0 Interrupt (Vector number: 37)</td>
</tr>
<tr>
<td>RX231</td>
<td></td>
</tr>
</tbody>
</table>

2.2.5 Header Files

All API calls and their supporting interface definitions are located in r_usb_basic_mini_if.h.
2.2.6 Integer Types

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

2.2.7 Compile Setting

For compile settings, refer to chapter 7, Configuration.

2.2.8 ROM / RAM Size

The follows show ROM/RAM size of this driver.

1. RX111, RX113

<table>
<thead>
<tr>
<th></th>
<th>Checks arguments</th>
<th>Does not check arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM size</td>
<td>15.3K bytes (Note 3)</td>
<td>15.1K bytes (Note 4)</td>
</tr>
<tr>
<td>RAM size</td>
<td>3.3K bytes</td>
<td>3.3K bytes</td>
</tr>
</tbody>
</table>

2. RX231

<table>
<thead>
<tr>
<th></th>
<th>Checks arguments</th>
<th>Does not check arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM size</td>
<td>15.5K bytes (Note 3)</td>
<td>15.3K bytes (Note 4)</td>
</tr>
<tr>
<td>RAM size</td>
<td>3.5K bytes</td>
<td>3.5K bytes</td>
</tr>
</tbody>
</table>

Note:
1. ROM/RAM size for BSP is included in the above size. The stack size is not included in the above RAM size.
2. The default option is specified in the compiler optimization 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_mini_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_mini_config.h file.

2.2.9 Argument

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

2.2.10 Adding the 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 e2 studio
By using the Smart Configurator in e2 studio, the FIT module is automatically added to your project. Refer to “Renesas e2 studio Smart Configurator User Guide (R20AN0451)” for details.

(2) Adding the FIT module to your project using the FIT Configurator in e2 studio
By using the FIT Configurator in e2 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 e2 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.

2.4 Class Request

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

2.5 Descriptor

2.5.1 String Descriptor

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

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

**Example descriptor construction**

```c
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.

```c
/* String Descriptor table */
uint8_t *smp_str_table[] = {
  smp_str_descriptor0,
  smp_str_descriptor1,
  smp_str_descriptor2,
};
```
RX Family

USB Basic Mini Host and Peripheral Driver (USB Mini Firmware) Using Firmware Integration Technology

```c
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 8.4, `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 8.4, `usb_descriptor_t` structure.

### 2.6 Peripheral Battery Charging (PBC)

This driver supports PBC.

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

You can get the result of CPD by calling `R_USB_GetInformation` function. For `R_USB_GetInformation` function, refer to chapter 4.12.
The processing flow of PBC is shown in Figure 2-1.
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.
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
### 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>
<thead>
<tr>
<th>Item</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrated Development Environment</td>
<td>Renesas Electronics e² studio V.7.0.0</td>
</tr>
<tr>
<td>C compiler</td>
<td>Renesas Electronics C/C++ compiler for RX Family V.2.08.00</td>
</tr>
<tr>
<td>Endian</td>
<td>Compile Option : -lang = c99</td>
</tr>
<tr>
<td>Endian</td>
<td>Little Endian, Big Endian</td>
</tr>
<tr>
<td>USB Driver Revision Number</td>
<td>Rev.1.10</td>
</tr>
<tr>
<td>Using Board</td>
<td>Renesas Starter Kit for RX111</td>
</tr>
<tr>
<td></td>
<td>Renesas Starter Kit for RX113</td>
</tr>
<tr>
<td></td>
<td>Renesas Starter Kit for RX231</td>
</tr>
</tbody>
</table>

#### 3.3.4 Usage of Interrupt Vector

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

<table>
<thead>
<tr>
<th>Device</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX111</td>
<td>USB10 Interrupt (Vector number: 36) / USBR0 Interrupt (Vector number: 90)</td>
</tr>
<tr>
<td>RX113</td>
<td>USB D0FIFO0 Interrupt (Vector number: 36) / USB D1FIFO0 Interrupt (Vector number: 37)</td>
</tr>
<tr>
<td>RX231</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.3.5 Header Files

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

#### 3.3.6 Integer Types

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

#### 3.3.7 Compile Setting
3.3.8 ROM / RAM Size

The follows show ROM/RAM size of this driver.

1. RX111, RX113

<table>
<thead>
<tr>
<th>Checks arguments</th>
<th>Does not check arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM size</td>
<td>17.8K bytes (Note 3)</td>
</tr>
<tr>
<td>RAM size</td>
<td>4.2K bytes</td>
</tr>
<tr>
<td></td>
<td>17.6K bytes (Note 4)</td>
</tr>
<tr>
<td></td>
<td>4.2K bytes</td>
</tr>
</tbody>
</table>

2. RX231

<table>
<thead>
<tr>
<th>Checks arguments</th>
<th>Does not check arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM size</td>
<td>18K bytes (Note 3)</td>
</tr>
<tr>
<td>RAM size</td>
<td>4.4K bytes</td>
</tr>
<tr>
<td></td>
<td>17.8K bytes (Note 4)</td>
</tr>
<tr>
<td></td>
<td>4.4K bytes</td>
</tr>
</tbody>
</table>

Note:
1. ROM/RAM size for BSP is included in the above size. The stack size is not included in the above RAM size.
2. The default option is specified in the compiler optimization 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_mini_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_mini_config.h file.

3.3.9 Argument

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

3.3.10 Adding the 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 e² studio

By using the Smart Configurator in e² studio, the FIT module is automatically added to your project. Refer to “Renesas e² studio Smart Configurator User Guide (R20AN0451)” for details.

(2) Adding the FIT module to your project using the FIT Configurator in e² studio

By using the FIT Configurator in e² 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 e² 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 chapter 9, USB Class Requests.

3.6 How to Set the Target Peripheral List (TPL)
By registering the Vendor ID (VID) and Product ID (PID) in the USB host, USB communication will only be enabled for the USB device identified with a registered VID and PID.

To register a USB device in the TPL, specify the VID and PID as a set to the macro definitions listed in Table 3-3 in the configuration file (r_usb_basic_mini_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>
<thead>
<tr>
<th>Macro definition name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_TPL_CNT</td>
<td>Specify the number of USB devices to be supported.</td>
</tr>
<tr>
<td>USB_TPL</td>
<td>Specify a VID/PID set for each USB device to be supported. (Always specify in the order of VID first, PID second.)</td>
</tr>
</tbody>
</table>

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

```c
#define USB_CFG_TPL   0x0011, 0x0022, 0x0033, 0x0044, 0x0055, 0x0066
#define USB_CFG_TPLCNT 3
```

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

```c
#define USB_CFG_TPLCNT 3
#define USB_CFG_TPL 0x0011, 0x0022, 0x0033, 0x0044, 0x0055, 0x0066
```

**Example 2) VID and PID registration not required**

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

Note:
1. Set USB_CFG_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_mini_config.h), refer to chapter 7, Configuration (r_usb_basic_mini_config.h).

3.7 Allocation of Device Addresses
In USB Host mode, the USB driver allocates device address value 1 to the connected USB devices.

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

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

You can get the result of Change Port Detection (CPD) by calling \textit{R_USB_GetInformation} function.
The processing flow of HBC is shown Figure 3-1.

![Diagram](image)

**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>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_USB_Open() (Note1)</td>
<td>Start the USB module</td>
</tr>
<tr>
<td>R_USB_Close() (Note1)</td>
<td>Stop the USB module</td>
</tr>
<tr>
<td>R_USB_GetVersion()</td>
<td>Get the driver version</td>
</tr>
<tr>
<td>R_USB_Read() (Note1)</td>
<td>Request USB data read</td>
</tr>
<tr>
<td>R_USB_Write() (Note1)</td>
<td>Request USB data write</td>
</tr>
<tr>
<td>R_USB_Stop() (Note1)</td>
<td>Stop USB data read/write processing</td>
</tr>
<tr>
<td>R_USB_Suspend() (Note1)</td>
<td>Request suspend</td>
</tr>
<tr>
<td>R_USB_Resume() (Note1)</td>
<td>Request resume</td>
</tr>
<tr>
<td>R_USB_GetEvent() (Note1)</td>
<td>Return USB-related completed events</td>
</tr>
<tr>
<td>R_USB_VbusSetting() (Note1)</td>
<td>Sets VBUS supply start/stop.</td>
</tr>
<tr>
<td>R_USB_PullUp() (Note1)</td>
<td>Pull-up enable/disable setting of D+/D- line</td>
</tr>
<tr>
<td>R_USB_GetInformation()</td>
<td>Get information on USB device.</td>
</tr>
<tr>
<td>R_USB_PipeRead() (Note1)</td>
<td>Request data read from specified pipe</td>
</tr>
<tr>
<td>R_USB_PipeWrite() (Note1)</td>
<td>Request data write to specified pipe</td>
</tr>
<tr>
<td>R_USB_PipeStop() (Note1)</td>
<td>Stop USB data read/write processing to specified pipe</td>
</tr>
<tr>
<td>R_USB_GetUsePipe()</td>
<td>Get pipe number</td>
</tr>
<tr>
<td>R_USB_GetPipeInfo()</td>
<td>Get pipe information</td>
</tr>
</tbody>
</table>

Note:

1. If the API of (Note 1) is executed on the same USB module by interrupt handling etc while the API of (Note 1) is executing, this USB driver may not work properly.

2. The class-specific API function other than the above API is supported in Host Mass Storage Class. Refer to the document (Document number: R01AN2169) for the class-specific API.

3. The class-specific API function other than the above API is supported in Host Human Interface Device Class. Refer to the document (Document number: R01AN2168) for the class-specific API.

4. 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 7, Configuration for USB_CFG_PARAM_CHECKING definition.
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

```c
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`.

Reentrant

This API is not reentrant.

Note

1. For details concerning the `usb_ctrl_t` structure, see chapter 8.1, `usb_ctrl_t structure`, and for the `usb_cfg_t` structure, see chapter 8.3, `usb_cfg_t structure`.
2. Assign the device class type (see chapter 6, Device Class Types) to the member (type) of the `usb_ctrl_t` structure. Does not assign `USB_HCDC` and `USB_PCD` to this member (type). If `USB_HCDCC` or `USB_PCDCC` is assigned, then `USB_ERR_PARA` will be returned.
3. 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.
4. 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.
5. If 0 (zero) is assigned to one of the arguments, then `USB_ERR_PARA` will be the return value.
Examples

1. In the case of USB Host mode

```c
void usb_host_application(void)
{
    usb_err_t.err;
    usb_ctrl_t.ctrl;
    usb_cfg_t.cfg;

    ctrl.type = USB_HCDC;
    cfg.usb_mode = USB_HOST;
    err = R_USB_Open(&ctrl, &cfg);  // Start USB module */
    if (USB_SUCCESS != err)
    {
        
    }
}
```

2. In the case of USB Peripheral

```c
usb_descriptor_t smp_descriptor =
{
    g_device,
    g_config_f,
    g_qualifier,
    g_string
};
void usb_peri_application(void)
{
    usb_err_t.err;
    usb_ctrl_t.ctrl;
    usb_cfg_t.cfg;

    ctrl.type = USB_PCDC;
    cfg.usb_mode = USB_PERI;
    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(void)

Arguments

    -

Return Value

<table>
<thead>
<tr>
<th>Success</th>
<th>USB_SUCCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB module is not open.</td>
<td>USB_ERR_NOT_OPEN</td>
</tr>
</tbody>
</table>

Description

This function terminates power to the USB module.

Reentrant

This API is not reentrant.

Note

- 

Example

```c
void    usr_application(void)
{
    usb_err_t    err;
    usb_ctrl_t   ctrl;

    err = R_USB_Close();
    if (USB_SUCCESS != err)
    {
        
    }
}
```
4.3 R_USB_GetVersion

Return API version number

Format

\[
\text{usb_err_t} \quad \text{R_USB_GetVersion()}
\]

Arguments

-

Return Value

Version number

Description

The version number of the USB driver is returned.

Reentrant

This API is reentrant.

Note

-

Example

\[
\text{void } \text{usr_application( void )} \\
\{ \\
\quad \text{uint32_t } \text{version}; \\
\quad : \\
\quad \text{version} = \text{R_USB_GetVersion();} \\
\quad : \\
\}
\]
4.4 R_USB_Read

USB data read request

Format

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

Arguments

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

Return Value

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

Description

1. Bulk/interrupt data transfer

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

2. Control data transfer

   Refer to chapter 9, USB Class Requests for details.

Reentrant

This API is not reentrant.

Note

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

2. When `USB_SUCCESS` is returned for the return value, it only means that a data read request was performed to
   the USB driver, not that the data read processing has completed. The completion of the data read can be
   checked by reading the return value (`USB_STS_READ_COMPLETE`) of the `R_USB_GetEvent` function.

3. When the read data is n times the maximum packet size and does not meet the read request size, the USB
   driver assumes the data transfer is still in process and `USB_STS_READ_COMPLETE` is not set as the return
   value of the `R_USB_GetEvent` function.

4. Before calling this API, assign the device class type (see chapter 6, Device Class Types) to the member (`type`)
   of the `usb_ctrl_t` structure.

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

6. The size of area assigned to the second argument (`p_buf`) must be at least as large as the size specified for the
   third argument (`size`). Allocate the area n times the max packet size when using DTC/DMA transfer.

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

8. In USB Host mode it is not possible to repeatedly call the `R_USB_Read` function. 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, first check the `USB_STS_READ_COMPLETE` return value from the `R_USB_GetEvent` function,
   and then call the `R_USB_Read` function.
9. 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.

10. In Vendor Class, use the `R_USB_PipeRead` function.

11. 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.

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

13. In the USB device is in the CONFIGURED state, this API can be called. If this API is called when the USB device is in other than the CONFIGURED state, then `USB_ERR_NG` will be the return value.

Example

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

USB data write request

Format

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

Arguments

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

Return Value

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

Description

1. Bulk/Interrupt data transfer

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

2. Control data transfer

   Refer to chapter 9, USB Class Requests for details.

Reentrant

This API is not reentrant.

Note

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

2. When `USB_SUCCESS` is returned for the return value, it only means that a data write request was performed to the USB driver, not that the data write processing has completed. The completion of the data write can be checked by reading the return value (`USB_STS_WRITE_COMPLETE`) of the `R_USB_GetEvent` function.

3. Before calling this API, assign the device class type (see chapter 6, Device Class Types) to the member (`type`) of the `usb_ctrl_t` structure.

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

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

6. 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.

7. 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, first check the `USB_STS_WRITE_COMPLETE` return value from the `R_USB_GetEvent` function, and then call the `R_USB_Write` function.

8. 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_ctrl_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.
9. In Vendor Class, use the `R_USB_PipeWrite` function.

10. If this API is called after assigning `USB_HCDC`, `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 Host Mass Storage Class, to access storage media, use the FAT (File Allocation Table) API rather than this API.

12. 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**

```c
void usb_application( void )
{
    usb_ctrl_t ctrl;

    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
        case USB_STS_READ_COMPLETE:
            ctrl.type = USB_HCDC;
            R_USB_Write(&ctrl, g_buf, size);
            break;
        case USB_STS_WRITE_COMPLETE:
            break;
        }
    }
}
```
4.6 R_USB_Stop

USB data read/write stop request

Format

```c
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`).

Reentrant

This API is not reentrant.

Note

1. Before calling this API, assign the device class type to the member (`type`) of the `usb_ctrl_t` structure.
2. If `USB_NULL` is assigned to the argument (`p_ctrl`), then `USB_ERR_PARA` will be the return value.
3. 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.
4. If `USB_HCDC` is assigned to the member (`type`) and `USB_WRITE` is assigned to the 2nd argument (`type`), then `USB_ERR_PARA` will be the return value.
5. If `USB_PCDCC` is assigned to the member (`type`) and `USB_READ` is assigned to the 2nd argument (`type`), then `USB_ERR_PARA` will be the return value.
6. In USB Host mode, `USB_ERR_NG` will be the return value when this API can not stop the data read/write request.
7. 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.
8. If this API is called after assigning `USB_HMSC`, `USB_PMSC`, `USB_HVND` or `USB_PVND` to the member (`type`) of the `usb_ctrl_t` structure, then `USB_ERR_PARA` will be the return value.
9. In Vendor Class, use the `R_USB_PipeStop` function.
10. Do not use this API for the Host Mass Storage Class.
11. 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

```c
void usb_application( void )
{
    usb_ctrl_t ctrl;

    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
            case USB_STS_DETACH:
            {
                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

```c
usb_err_t R_USB_Suspend(void)
```

Arguments

---

Return Value

- **USB_SUCCESS**: Successfully completed
- **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

Sends a SUSPEND signal from the USB module.

After the suspend request is completed, confirm the operation with the return value (USB_STS_SUSPEND) of the R_USB_GetEvent function.

Reentrant

This API is not reentrant.

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

```c
void usb_host_application( void )
{
    usb_ctrl_t ctrl;
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
            case USB_STS_NONE:
                R_USB_Suspend();
                break;
            case USB_STS_SUSPEND:
                break;
        }
    }
}
```
4.8 R_USB_Resume

Resume signal transmission

Format

```
usb_err_t   R_USB_Resume(void)
```

Arguments

---

Return Value

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_SUCCESS</td>
<td>Successfully completed</td>
</tr>
<tr>
<td>USB_ERR_BUSY</td>
<td>Resume already requested for same device address (USB host mode only)</td>
</tr>
<tr>
<td>USB_ERR_NOT_SUSPEND</td>
<td>USB device is not in the SUSPEND state.</td>
</tr>
<tr>
<td>USB_ERR_NG</td>
<td>USB device is not the state to be able to request the remote wakeup (USB peripheral mode only)</td>
</tr>
</tbody>
</table>

Description

This function sends a RESUME signal from the USB module.

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

Reentrant

This API is not reentrant.

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 command 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. 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.
Example

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

Get completed USB-related events

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)`.

Reentrant

This API is not reentrant.

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 5, *Return Value of R_USB_GetEvent Function* for details on the completed event value used as the API return value.
3. If there is no completed event when calling this API, then `USB_STS_NONE` will be the return value.
4. Please call this API in the main loop of the user application program.

Example

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

VBUS Supply Start/Stop Specification

Format

```c
usb_err_t R_USB_VbusSetting(uint16_t state)
```

Arguments

- `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.

Reentrant

This API is not reentrant.

Note

1. For information on setting the VBUS output of the power source IC for the USB Host to either Low Assert or High Assert, see the setting of the `USB_CFG_VBUS` definition described in chapter 7, Configuration (`r_usb_basic_mini_config.h`).
2. 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 argument, this driver operates the same processing as when `USB_OFF` is set.
3. Use this API only when need the control of VBUS in the application program. (This driver does not control VBUS after turning on VBUS in the initialization processing.)
4. This API is processed only in USB Host mode. If this API is called in USB Peripheral mode, then `USB_ERR_NG` will be the return value.

Example

```c
void usb_host_application( void )
{
    usb_ctrl_t ctrl;
    :
    :
    R_USB_VbusSetting( &ctrl, USB_ON ); /* Start VBUS supply */
    :
    :
    R_USB_VbusSetting( &ctrl, USB_OFF ); /* Stop VBUS supply */
    :
    :
}
```
4.11  R_USB_PullUp

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

Format

```c
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.

Reentrant

This API is not reentrant.

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.

Example

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

}
4.12 R_USB_GetInformation

Get USB device information

Format

```
usb_err_t
R_USB_GetInformation(usb_info_t *p_info)
```

Arguments

- `p_info` Pointer to `usb_info_t` structure area

Return Value

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

Description

This function obtains completed USB-related events.

For information to be obtained, see chapter 8.6, `usb_info_t` structure.

Reentrant

This API is reentrant.

Note

1. Call this API after calling the `R_USB_Open` function (and before calling the `R_USB_Close` function). `USB_ERR_NG` will be the return value when calling this API before calling `R_USB_Open` function.
2. Do not assign `USB_NULL` to the second argument (`p_info`). If `USB_NULL` is assigned, then `USB_ERR_PARA` will be the return value.

Example

```c
void usb_host_application( void )
{
    usb_info_t info;
    :
    R_USB_GetInformation(&info);
    :
}
```
4.13 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_NG`: Other error

Description

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

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

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

Reentrant

This API is reentrant for different USB PIPE

Note

1. This API only performs data read request processing. An application program does not wait for data read completion by using this API.
2. When `USB_SUCCESS` is returned for the return value, it only means that a data read request was performed to the USB driver, not that the data read processing has completed. The completion of the data read can be checked by reading the return value (`USB_STS_READ_COMPLETE`) of the `R_USB_GetEvent` function.
3. When the read data is n times the max packet size and does not meet the read request size, the USB driver assumes the data transfer is still in process and `USB_STS_READ_COMPLETE` is not set as the return value of the `R_USB_GetEvent` function.
4. Before calling this API, assign the PIPE number (USB_PIPE1 to USB_PIPE9) to be used to the member (`pipe`) of the `usb_ctrl_t` structure.
5. If something other than USB_PIPE1 through USB_PIPE9 is assigned to the member (`pipe`) of the `usb_ctrl_t` structure, then `USB_ERR_PARA` will be the return value.
6. Do not assign a pointer to the auto variable (stack) area to the second argument (`p_buf`).
7. The size of area assigned to the second argument (`p_buf`) must be at least as large as the size specified for the third argument (`size`). Allocate the area n times the max packet size when using DTC/DMA transfer.
8. If 0 (zero) is assigned to one of the arguments, then `USB_ERR_PARA` will be the return value.
9. It is not possible to repeatedly call the `R_USB_PipeRead` function with the same value assigned to the member (`pipe`) of the `usb_ctrl_t` structure. If the `R_USB_PipeRead` function is called repeatedly, then `USB_ERR_BUSY` will be the return value. To call the `R_USB_PipeRead` function more than once with the same value assigned to the member (`pipe`), first check the `USB_STS_READ_COMPLETE` return value from the `R_USB_GetEvent` function, and then call the `R_USB_PipeRead` function.
11. In CDC/HID Class, to perform a Bulk/Interrupt transfer, use the `R_USB_Read` function rather than this API. With Host Mass Storage Class, to perform data access to the MSC device, use the FAT (File Allocation Table) API rather than this API.

12. Assign nothing to the member (type) of the `usb_ctrl_t` structure. Even if the device class type or something is assigned to the member (type), it is ignored.

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

14. 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 7, Configuration.

15. 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**

```c
void usb_application( void )
{
    usb_ctrl_t ctrl;
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
            case USB_STS_WRITE_COMPLETE:
            :
                ctrl.pipe = USB_PIPE1;
                R_USB_PipeRead(&ctrl, buf, size);
                break;
            case USB_STS_READ_COMPLETE:
            :
                break;
        }
    }
}
```
4.14 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`: Specified pipe now handling data receive/send request
- `USB_ERR_NG`: Other error

Description

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

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

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

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

Reentrant

This API is reentrant for different USB PIPE

Note

1. This API only performs data write request processing. An application program does not wait for data write completion by using this API.
2. When `USB_SUCCESS` is returned for the return value, it only means that a data write request was performed to the USB driver, not that the data write processing has completed. The completion of the data write can be checked by reading the return value (`USB_STS_WRITE_COMPLETE`) of the `R_USB_GetEvent` function.
3. Before calling this API, assign the PIPE number (`USB_PIPE1` to `USB_PIPE9`) to be used to the member (`pipe`) of the `usb_ctrl_t` structure.
4. 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.
5. Do not assign a pointer to the auto variable (stack) area to the second argument (`p_buf`).
6. If 0 (zero) is assigned to the argument (`p_ctrl` or `p_buf`), then `USB_ERR_PARA` will be the return value.
7. It is not possible to repeatedly call the `R_USB_PipeWrite` function with the same value assigned to the member (`pipe`) of the `usb_ctrl_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.
8. 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.
9. 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.
10. To transfer the data for a Control transfer, use the \texttt{R_USB_Write} function rather than this API.

11. Enable one of \texttt{USB_CFG_HVND_USB} or \texttt{USB_CFG_PVND_USE} definition when using this API. If this API is used when these definitions are not enabled, \texttt{USB_ERR_NG} is returned. For \texttt{USB_CFG_HVND_USB} or \texttt{USB_CFG_PVND_USE} definition, refer to chapter 7, \textit{Configuration}.

12. This API can be called when the USB device is in the configured state. When the API is called in any other state, \texttt{USB_ERR_NG} is returned.

\textbf{Example}

```c
void usb_application( void )
{
    usb_ctrl_t ctrl;
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
            case USB_STS_READ_COMPLETE:
            {
                ctrl.pipe = USB_PIPE2;
                R_USB_PipeWrite(&ctrl, g_buf, size);
                break;
            }
            case USB_STS_WRITE_COMPLETE:
            {
                break;
            }
        }
    }
}
```
4.15 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

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_SUCCESS</td>
<td>Successfully completed (stop request completed)</td>
</tr>
<tr>
<td>USB_ERR_PARA</td>
<td>Parameter error</td>
</tr>
<tr>
<td>USB_ERR_NG</td>
<td>Other error</td>
</tr>
</tbody>
</table>

Description

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

Reentrant

This API is reentrant for different USB PIPE

Note

1. Before calling this API, specify the selected pipe number (`USB_PIPE0` to `USB_PIPE9`) in the `usb_ctrl_t` member (`pipe`).
2. 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.
3. `USB_ERR_PARA` will be the return value when `USB_NULL` is assigned to the argument (`p_ctrl`).
4. In USB Host mode, `USB_ERR_NG` will be the return value when this API can not stop the data read/write request.
5. 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.
6. 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.
7. 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 7, Configuration.
8. This API can be called when the USB device is in the configured state. When the API is called in any other state, `USB_ERR_NG` is returned.
Example

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

Get used pipe number from bit map

Format

```c
usb_err_t R_USB_GetUsePipe(uint16_t *p_pipe)
```

Arguments

- `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 initialization) via bit map information. The bit map information is stored in the area specified in argument (`p_pipe`).

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

```
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`).

Reentrant

This API is reentrant.

Note

1. Bit map information b0(PIPE0) is always set to "1".
2. `USB_ERR_PARA` will be the return value when `USB_NULL` is assigned to the argument (`p_pipe`).
3. 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

```c
void usb_application( void )
{
    uint16_t usepipe;
    usb_ctrl_t ctrl;
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
            case USB_STS_CONFIGURED:
                R_USB_GetUsePipe(&ctrl, &usepipe);
                break;
        }
    }
}
```
4.17  R_USB_GetPipeInfo

Get pipe information for specified pipe

Format

```c
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`).

Reentrant

This API is reentrant.

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. If 0 (zero) is assigned to one of the arguments, then `USB_ERR_PARA` will be the return value.
3. Refer to chapter 8.5, `usb_pipe_t structure` for details on the `usb_pipe_t` structure.
4. 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

```c
void usb_host_application( void )
{
    usb_pipe_t info;
    usb_ctrl_t ctrl;
    while (1)
    {
        switch (R_USB_GetEvent(&ctrl))
        {
            case USB_STS_CONFIGURED:
                ctrl.pipe = USB_PIPE3;
                R_USB_GetPipeInfo( &ctrl, &info );
                break;
        }
    }
}
```
5. Return Value of R_USB_GetEvent Function

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

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

5.1 USB_STS_DEFAULT

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

5.2 USB_STS_CONFIGURED

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

5.3 USB_STS_SUSPEND

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

5.4 USB_STS_RESUME

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

Note:

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

5.5 USB_STS_DETACH

When the R_USB_GetEvent function is called after the USB device has been detached from the USB host, the function sends USB_STS_DETACH as the return value.
5.6 USB_STS_REQUEST

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

- \texttt{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 \texttt{R_USB_GetEvent} function is called, \texttt{USB_STS_REQUEST_COMPLETE} is sent as the return value instead of \texttt{USB_STS_REQUEST}.
2. For more details on USB request information (8 bytes) stored in member \texttt{(setup)}, refer to chapter 8.2, \texttt{usb_setup_t} structure.

5.7 USB_STS_REQUEST_COMPLETE

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

- \texttt{status} : Sets either USB_ACK / USBSTALL

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 \texttt{usb_ctrl_t} structure member \texttt{(setup)}. For more details on USB request information (8 bytes) stored in member \texttt{(setup)}, refer to chapter 8.2, \texttt{usb_setup_t} structure.

5.8 USB_STS_READ_COMPLETE

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

- \texttt{type} : Device class type of completed data read (only set when using \texttt{R_USB_Read} function)
- \texttt{size} : Size of read data
- \texttt{pipe} : Pipe number of completed data read
- \texttt{status} : Read completion error information

Note:

1. In the case of the \texttt{R_USB_PipeRead} function, the member \texttt{(pipe)} has the PIPE number (\texttt{USB_PIPE1} to \texttt{USB_PIPE9}) for which data read is completed. In the case of the \texttt{R_USB_Read} function, \texttt{USB_NULL} is set to the member \texttt{(pipe)}.
2. For details on device class type, refer to chapter 6, Device Class Types.
3. The member \texttt{(status)} has the read completion error information. The error information set to this member is as follows.

- \texttt{USB_SUCCESS} : Data read successfully completed
- \texttt{USB_ERR_OVER} : Received data size over
- \texttt{USB_ERR_SHORT} : Received data size short
- \texttt{USB_ERR_NG} : Data reception failed

(1). Even if the reception request size is less than MaxPacketSize \times n, if MaxPacketSize \times n bytes of data are received, then \texttt{USB_ERR_OVER} is set.

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

(2). If the reception request size is less than MaxPacketSize \times n and the actual received data size is less than this reception request size, then \texttt{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 \texttt{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`.

5.9 USB_STS_WRITE_COMPLETE

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

- **type**: Device class type of completed data write (only set when using `R_USB_Write` function)
- **pipe**: Pipe number of completed data write
- **status**: Write completion error information

Note:

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

   - `USB_SUCCESS`: Data write successfully completed
   - `USB_ERR_NG`: Data transmission failed

5.10 USB_STS_BC

If the `R_USB_GetEvent` function is called after connecting to the USB device/USB Host that supports the Battery Charging function is detected, then `USB_STS_BC` will be the return value.

5.11 USB_STS_OVERCURRENT

In USB Host mode, if the `R_USB_GetEvent` function is called after overcurrent is detected, then `USB_STS_OVERCURRENT` will be the return value.

5.12 USB_STS_NOT_SUPPORT

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

5.13 USB_STS_NONE

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

- **status**: USB device status
6. Device Class Types

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

<table>
<thead>
<tr>
<th>Device class type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_HCDC</td>
<td>Host Communication Device Class</td>
</tr>
<tr>
<td>USB_HCDCC</td>
<td>Host Communication Device Class (Control Class)</td>
</tr>
<tr>
<td>USB_HHID</td>
<td>Host Human Interface Device Class</td>
</tr>
<tr>
<td>USB_HMSC</td>
<td>Host Mass Storage Device Class</td>
</tr>
<tr>
<td>USB_PCDC</td>
<td>Peripheral Communication Device Class</td>
</tr>
<tr>
<td>USB_PCDCC</td>
<td>Peripheral Communication Device Class (Control Class)</td>
</tr>
<tr>
<td>USB_PHID</td>
<td>Peripheral Human Interface Device Class</td>
</tr>
<tr>
<td>USB_PMSC</td>
<td>Peripheral Mass Storage Device Class</td>
</tr>
<tr>
<td>USB_HVNDR</td>
<td>Host Vendor Class</td>
</tr>
<tr>
<td>USB_PVNDR</td>
<td>Peripheral Vendor Class</td>
</tr>
</tbody>
</table>

Note:

1. Host Communication Device Class: When transmitting data in a bulk transfer, specify *USB_HCDC* in the *usb_ctrl_t* structure member (type). When transmitting data in an interrupt transfer, specify *USB_HCDC* in the *usb_ctrl_t* structure member (type).

2. Peripheral Communication Device Class: When transmitting data in a bulk transfer, specify *USB_PCDC* in the *usb_ctrl_t* structure member (type). When transmitting data in an interrupt transfer, specify *USB_PCDCC* in the *usb_ctrl_t* structure member (type).

3. For an application program, do not assign *USB_HMSC*, *USB_PMSC*, *USB_HVND*, and *USB_PVND* to the member (type) of the *usb_ctrl_t* structure.
7. Configuration (r_usb_basic_mini_config.h)

7.1 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 \texttt{USB\_CFG\_MODE}.

   (1). USB Host mode

   Set \texttt{USB\_CFG\_HOST} for the definition of \texttt{USB\_CFG\_MODE}.
   
   \begin{verbatim}
   #define USB_CFG_MODE USB_CFG_HOST
   \end{verbatim}

   (2). USB Peripheral mode

   Set \texttt{USB\_CFG\_PERI} for the definition of \texttt{USB\_CFG\_MODE}.
   
   \begin{verbatim}
   #define USB_CFG_MODE USB_CFG_PERI
   \end{verbatim}

2. Argument check setting

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

   \begin{verbatim}
   #define USB_CFG_PARAM_CHECKING USB_CFG_ENABLE \quad // Checks arguments.
   #define USB_CFG_PARAM_CHECKING USB_CFG_DISABLE \quad // Does not check arguments.
   \end{verbatim}

3. Device class setting

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

   \begin{verbatim}
   #define USB_CFG_HCDC_USE \quad // Host Communication Device Class
   #define USB_CFG_HHID_USE \quad // Host Human Interface Device Class
   #define USB_CFG_HMSC_USE \quad // Host Mass Storage Class
   #define USB_CFG_HVNDR_USE \quad // Host Vendor Class
   #define USB_CFG_PCDC_USE \quad // Peripheral Communication Device Class
   #define USB_CFG_PHID_USE \quad // Peripheral Human Interface Device Class
   #define USB_CFG_PMSC_USE \quad // Peripheral Mass Storage Class
   #define USB_CFG_PVNDR_USE \quad // Peripheral Vendor Class
   \end{verbatim}

4. DTC use setting

Specify whether to use the DTC.

   \begin{verbatim}
   #define USB_CFG_DTC USB_CFG_ENABLE \quad // Uses DTC
   #define USB_CFG_DTC USB_CFG_DISABLE \quad // Does not use DTC
   \end{verbatim}

Note:

If \texttt{USB\_CFG\_ENABLE} is set for the definition of \texttt{USB\_CFG\_DTC}, be sure to set \texttt{USB\_CFG\_DISABLE} for the definition of \texttt{USB\_CFG\_DMA} in 5 below.

5. DMA use setting

Specify whether to use the DMA.

   \begin{verbatim}
   #define USB_CFG_DMA USB_CFG_ENABLE \quad // Uses DMA.
   #define USB_CFG_DMA USB_CFG_DISABLE \quad // Does not use DMA.
   \end{verbatim}

Note:

   (1). Be sure to specify \texttt{USB\_CFG\_DISABLE} when using RX111/RX113.

   (2). If \texttt{USB\_CFG\_ENABLE} is set for the definition of \texttt{USB\_CFG\_DMA}, be sure to set \texttt{USB\_CFG\_DISABLE} for the definition of \texttt{USB\_CFG\_DTC} in 4 above.

   (3). If \texttt{USB\_CFG\_ENABLE} is set for the definition of \texttt{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

Note:

(1). Set one of the DMA channel numbers from USB_CFG_CH0 to USB_CFG_CH3. 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 using USB Host Mass Storage class.

The following is the specifying example.

a. When using the DMA transfer for DMA sending and receiving

#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 USB0 module

#define USB_CFG_USB0_DMA_TX USB_CFG_CH1

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.

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.

8. 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)

9. USB regulator setting

Specify whether your system uses USB regulator function supported by RX231 or not.

#define USB_CFG_REGULATOR USB_CFG_OFF // No use
#define USB_CFG_REGULATOR USB_CFG_ON // Use

Note:
This definition is ignored when using MCU except RX231.

7.2 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_DISABLE is set for this definition, then set USB_CFG_ENABLE for the definition of USB_CFG_BC in above.

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

   #define USB_CFG_COMPLIANCE_USB_CFG_ENABLE  // Compliance Test supported.
   #define USB_CFG_COMPLIANCE_USB_CFG_DISABLE  // Compliance Test not supported.

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

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

7.3 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. Request notification setting
   Set whether this driver notifies the application program the reception of SET_INTERFACE, SET_FEATURE/CLEARFEATURE 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.

7.4 Other Definitions
   In addition to the above, the following definition is also provided in r_usb_basic_mini_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.

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

This chapter describes the structures used in the application program.

8.1 usb_ctrl_t structure

The usb_ctrl_t structure is used for USB data transmission and other operations.

```c
typedef struct usb_ctrl {
    uint8_t pipe; /* Note 1 */
    uint8_t type; /* Note 2 */
    uint16_t status; /* Note 3 */
    uint32_t size; /* Note 4 */
    usb_set_up setup; /* Note 5 */
} 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 this member. Therefore, except when initializing the usb_ctrl_t structure area or processing an ACK/STALL response to a vendor class request, the application program should not write into this member.
6. Member (size) is used to set the size of data that is read. The USB driver sets this member. Therefore, the application program should not write into this member.
7. Member (setup) is used to set the information about a class request.

8.2 usb_setup_t structure

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

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

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

8.3 usb_cfg_t structure

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

```c
typedef struct usb_cfg {
    uint8_t usb_mode;  /* Note 1 */
    usb_descriptor_t *p_usb_reg;  /* Note 2 */
} 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_descriptor_t type pointer for the USB device in member (p_usb_reg). Refer to chapter 8.4, usb_descriptor_t structure for details on the usb_descriptor_t type. This member can only be set in USB peripheral mode. Even if it is set in USB host mode, the settings will be ignored.

8.4 usb_descriptor_t structure

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

```c
typedef struct usb_descriptor {
    uint8_t *p_device;  /* Note 1 */
    uint8_t *p_config_f;  /* Note 2 */
    uint8_t **p_p_string;  /* Note 3 */
    uint8_t num_string;  /* Note 4 */
} 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).
3. 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.

```c
usb_descriptor_t usb_descriptor =
{
    smp_device,
    smp_config_f,
    smp_string,
    3,
};
```
4. Specify the number of the string descriptor which set in the string descriptor table to the member (num_string).

8.5 usbd_pipe_t structure

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

```c
typedef struct usbd_pipe {
    uint8_t ep; /* Note 1 */
    uint8_t type; /* Note 2 */
    uint16_t mxps; /* Note 3 */
} usbd_pipe_t;
```

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

8.6 usbd_info_t structure

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

```c
typedef struct usbd_info {
    uint8_t type; /* Note 1 */
    uint8_t speed; /* Note 2 */
    uint8_t status; /* Note 3 */
    uint8_t port; /* Note 4 */
} usbd_info_t;
```

Note:
1. In USB Host mode, the device class type of the connected USB device is set for the member (type). If the USB device is not connected, then USB_NOT_CONNECT is set. In USB Peripheral mode, the supporting device class type is set for the member (type). For information on the device class types, see 6, Device Class Types. (In the case of PCDC, USB_PCDC is set in this member(type))
2. The USB speed (USB_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_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 connected to the port is set to the member (port).
   - USB_SDP: Standard Downstream Port
   - USB_CDP: Charging Downstream Port
   - USB_DCP: Dedicated Charging Port (USB Peripheral only)

8.7 usbd_compliance_t structure

This structure is used when running the USB compliance test. The structure specifies the following USB-related information:
typedef struct usb_compliance {
    usb_ct_status_t status; /* Note 1 */
    uint16_t vid; /* Note 2 */
    uint16_t pid; /* Note 3 */
} usb_compliance_t;

Note:

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

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

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

3. The member pid is set to a value indicating the product ID of the connected USB device.
9. **USB Class Requests**

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

9.1 **USB Host operations**

9.1.1 **USB request (setup) transfer**

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

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

9.1.2 **USB request completion**

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

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

<table>
<thead>
<tr>
<th>status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_ACK</td>
<td>Successfully completed</td>
</tr>
<tr>
<td>USBSTALL</td>
<td>Stalled</td>
</tr>
</tbody>
</table>

9.1.3 **USB request processing example**

```c
void usr_application (void )
{
    usb_ctrl_t ctrl;
    while (1)
    {
        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;
                }
            
            break;
        }
    }
}
```

Nov 30, 2018
9.2 USB Peripheral operations

9.2.1 USB request (Setup)

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

9.2.2 USB request data

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

1. Receive procedure
   (1). Set the USB_REQUEST in the usb_ctrl_t structure member (type).
   (2). In the R_USB_Read function, specify the pointer to area that stores data in the second argument, and the requested data size in the third argument.
   (3). Call the R_USB_Read function.

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

2. Send procedure
   (1). Set USB_REQUEST in the usb_ctrl_t structure member (type).
   (2). Store the data from the data stage in a buffer. In the R_USB_Write function, specify the top address of the buffer in the second argument, and the transfer data size in the third argument.
   (3). Call the R_USB_Write function.

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

9.2.3 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 9.2.4, Example USB request processing description.

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

(2). Case of respondingSTALL to a class request

Note:
When receiving a class request that support the data stage, this USB driver process the status stage after processing the data stage.
9.2.4 Example USB request processing description

1. Request that supports control read data stage

```c
void usr_application (void )
{
    usb_ctrl_t ctrl;
    while (1)
    {
        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. Request that supports control write data stage

```c
void usr_application (void )
{
    usb_ctrl_t ctrl;
    while (1)
    {
        switch( R_USB_GetEvent( &ctrl ) )
        {
            case USB_REQUEST: /* Receive USB request */
                /* ctrl.setup analysis processing */
                ctrl.type = USB_REQUEST;
                R_USB_Read(&ctrl, g_buf, size); /* data (data stage) receive request */
                break;
            case USB_STS_REQUEST_COMPLETE:
                break;
        }
    }
}
3. Request that supports no-data control status stage (ACK response)

(1). ACK response

void usr_application (void )
{
    usb_ctrl_t ctrl;
    
    while (1) {
        switch( R_USB_GetEvent( &ctrl ) )
        {
            case USB_REQUEST: /* Receive USB 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

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

10.1 Basic Specification

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

Table10-1 DTC/DMA Setting Specifications

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO port used</td>
<td>D0FIFO and D1FIFO port</td>
</tr>
<tr>
<td>Transfer mode</td>
<td>Block transfer mode</td>
</tr>
<tr>
<td>Chain transfer</td>
<td>Disabled</td>
</tr>
<tr>
<td>Address mode</td>
<td>Full address mode</td>
</tr>
<tr>
<td>Read skip</td>
<td>Disabled</td>
</tr>
<tr>
<td>Access bit width (MBW)</td>
<td>2-byte transfer: 16-bit width</td>
</tr>
<tr>
<td>USB transfer type</td>
<td>BULK transfer</td>
</tr>
<tr>
<td>Transfer end</td>
<td>Receive direction: BRDY interrupt</td>
</tr>
<tr>
<td></td>
<td>Transmit direction: D0FIFO/D1FIFO interrupt, BEMP interrupt</td>
</tr>
</tbody>
</table>

Note:
This driver does not support using DMA transfer and DTC transfer at the same time.

10.2 Notes

10.2.1 DTC transfer

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

10.2.2 Data Reception Buffer Size

The user needs to allocate the buffer area for more than n times the max packet size to store the receiving data.

10.2.3 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.

10.2.4 Initialization Function for DMA/DTC transfer

Call the following DMA/DTC transfer initialization function in the user application program.

<table>
<thead>
<tr>
<th>Transfer Type</th>
<th>Initialization Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTC</td>
<td>R_DTC_Open</td>
</tr>
<tr>
<td>DMA</td>
<td>R_DMACA_Init</td>
</tr>
<tr>
<td></td>
<td>R_DMACA_Open</td>
</tr>
</tbody>
</table>

Note:
Specify the following definition to the argument in R_DMACA_Open function. For the detail of the following definition, refer to chapter 7, Configuration (r_usb_basic_mini_config.h).

```c
USB_CFG_USB0_DMA_TX, USB_CFG_USB0_DMA_RX
```

Example)

```c
R_DMACA_Init();
R_DMACA_Open(USB_CFG_USB0_DMA_TX);
R_DMACA_Open(USB_CFG_USB0_DMA_RX);
```
11. Additional Notes

11.1 Vendor ID

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

11.2 Compliance Test

In order to run the USB Compliance Test it is necessary to display USB device–related information on a display device such as an LCD. When the `USB_CFG_COMPLIANCE` definition in the configuration file (`r_usb_basic_mini_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 chapter 8.7, `usb_compliance_t structure`.
3. When the `USB_CFG_COMPLIANCE` definition in `r_usb_basic_mini_config.h` is set to `USB_CFG_ENABLE`, it is necessary to register the vendor ID and product ID in the TPL definition. For information on TPL definitions, refer to chapter 3.6, `How to Set the Target Peripheral List (TPL)`.
4. For a program sample of the `usb_compliance_disp` function, see 13.1, `usb_compliance_disp function`.

11.3 QE for USB

Copy the following file when using `QE for USB` V.1.2.1.

File Name : qe_usb_firm_setting.xml
Copy Source Folder : ProjectFolder/src/smc_gen/r_usb_basic_mini/utilities
Copy Destination Folder : ProjectFolder

Note:

The `ProjectFolder` means the folder where the `.cproject` file and the `.project` file are existed.
12. Creating an Application Program

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

12.1 Configuration

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

12.2 Descriptor Creation

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

12.3 Application Program Creation

12.3.1 Include

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

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

12.3.2 Initialization

1. USB 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.

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>I/O</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_VBUS</td>
<td>input</td>
<td>VBUS pin for USB communication</td>
</tr>
</tbody>
</table>

Table 12-2 USB I/O Pin Settings for USB Host Operation

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>I/O</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_VBUSEN</td>
<td>output</td>
<td>VBUS output enabled pin for USB communication</td>
</tr>
<tr>
<td>USB_OVRCURA</td>
<td>input</td>
<td>Overcurrent detection pin for USB communication</td>
</tr>
</tbody>
</table>

Note:
Please refer to the corresponding MCU user’s manual for the pin settings in ports used for your application program.

2. DTC/DMA-related initialization

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

<table>
<thead>
<tr>
<th>Transfer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTC</td>
<td>R_DTC_Open</td>
</tr>
<tr>
<td>DMA</td>
<td>R_DMACA_Init</td>
</tr>
<tr>
<td></td>
<td>R_DMACA_Open</td>
</tr>
</tbody>
</table>

Note:
(1). The setting for DTC/DMA transfer is needed when using DTC/DMA transfer. Refer to chapter 7, Configuration (r_usb_basic_mini_config.h).

(2). You need to specify the using DMA channel number to the argument for R_DMACA_Open function when using DMA transfer. Be sure to specify one of the following definitions for the argument.
Example 1) DMA transmission setting
R_DMACA_Open(USB_CFG_USB0_DMA_TX);
(Specify one of USB PIPE3 to USB PIPE5 for the reception USB pipe.)

Example 2) DMA reception setting
R_DMACA_Open(USB_CFG_USB0_DMA_RX);
(Specify one of USB PIPE3 to USB PIPE5 for the reception USB pipe.)

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

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

3. USB-related initialization

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

12.3.3 Descriptor Creation

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

12.3.4 Main routine

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

12.3.5 Application program description example (CPU transfer)

```c
#include "r_usb_basic_mini_if.h"
#include "r_usb_pcdc_mini_if.h"

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

    /* MCU pin setting */
    usb_pin_setting();

    /* Initialization processing */
    cfg.usb_mode = USB_PERI; /* Specify either USB host or USB peri */
    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:
                break;
            case USB_STS_WRITE_COMPLETE:
                ctrl.type = USB_PCDC;
                R_USB_Read( &ctrl.g_buf, 64 );
                break;
            default: break;
        }
    }
}
```
case USB_STS_READ_COMPLETE:
    ctrl.type = USB_PCDC;
    R_USB_Write( &ctrl, g_buf, ctrl.size );
    break;
default:
    break;
}
}

12.3.6 Application program description example (DMA transfer)
#include "r_usb_basic_mini_if.h"
#include "r_usb_pcdc_mini_if.h"

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

    /* MCU pin setting */
    usb_pin_setting();

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

    /* Initialization processing */
    cfg.usb_mode = USB_PERI; /* Specify either USB host or USB peri */
    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;
        }
    }
}

12.3.7 Application program description example (DTC transfer)
#include "r_usb_basic_mini_if.h"
#include "r_usb_pcdc_mini_if.h"

void usb_peri_application( void )
{

```c
usb_ctrl_t ctrl;
usb_cfg_t cfg;

/* MCU pin setting */
usb_pin_setting();

/* DTC initialization processing */
R_DTC_Open();

/* Initialization processing */
cfg.usb_mode = USB_PERI; /* Specify either USB host or USB peri */
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;
    }
}
```
13. Program Sample

13.1 usb_compliance_disp function

```c
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

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>Dec 1, 2014</td>
<td>—</td>
<td>First edition issued</td>
</tr>
<tr>
<td>1.01</td>
<td>Jun 1, 2015</td>
<td>—</td>
<td>RX231 is added in the target device</td>
</tr>
</tbody>
</table>
| 1.02 | Dec 28, 2015| —    | 1. "USB_REGULATOR" definition is added newly in "r_usb_basic_mini_config.h" file.  
|      |            |      | 2. In Host mode, USB driver is changed so that the null packet is not sent when the transmission data size of the control transfer is the max packet size \( \times n \). |
|      |            |      | 2. Supporting DMA transfer                                                  |
|      |            |      | 3. The following chapters are added.                                         
|      |            |      | (1) 3.7 Allocation of Device Addresses                                        |
|      |            |      | (2) 5 Return Value of R_USB_GetEvent Function                                |
|      |            |      | (3) 6 Device Class Types                                                     |
|      |            |      | (4) 11 Additional Notes                                                      |
|      |            |      | (5) 12 Creating an Application Program                                       |
|      |            |      | (6) 13 Program Sample                                                        |
|      |            |      | 4. The following chapters are changed.                                       
|      |            |      | (1) 3.6 How to Set the Target Peripheral List (TPL)                         |
|      |            |      | (2) 4 API Functions                                                          |
|      |            |      | (3) 7 Configuration (r_usb_basic_mini_config.h)                             |
|      |            |      | (4) 8 Structures                                                             |
|      |            |      | (5) 9 USB Class Requests                                                     |
|      |            |      | (6) 10 DTC/DMA Transfer                                                      |
|      |            |      | 5. The following chapters are deleted.                                       
|      |            |      | "How to Register Class Driver", "Task ID and Priority Setting", "Pipe Information Table", "Scheduler" |

---

**A-1**
General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

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

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

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

3. Prohibition of Access to Reserved Addresses
   Access to reserved addresses is prohibited.
   ⎯ The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals
   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.
   ⎯ When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

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

5. 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.

6. 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 implantation; etc.), or may cause serious property damage (space craft; underwater 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. 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.

8. 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.

9. 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.

10. 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.

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

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

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

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