RZ/T1 Group

R01AN2635EJ0130
Rev.1.30
Dec 07, 2017

USB Host Communications Device Class Driver (HCDC)

Introduction

This application note describes USB Host Communication Device Class Driver (HCDC). This module performs hardware control of USB communication. It is referred to below as the USB-BASIC-F/W.

The sample program of this application note is created based on "RZ/T1 group Initial Settings Rev.1.30". Please refer to "RZ/T1 group Initial Settings application note (R01AN2554EJ0130)" about operating environment.

Target Device

RZ/T1 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
2. USB Class Definitions for Communications Devices Revision 1.2
3. USB Communications Class Subclass Specification for PSTN Devices Revision 1.2
   http://www.usb.org/developers/docs/
4. RZ/T1 Group User's Manual: Hardware (Document No.R01UH0483EJ0130)
5. RZ/T1 Group Initial Settings (Document No.R01AN2554EJ0130)
6. USB Host Basic Firmware (Document No.R01AN2633EJ0130)

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

Contents

1. Overview

The USB HCDC, when used in combination with the USB-BASIC-F/W, operates as a USB host communications device class driver. The HCDC conforms to the PSTN device subclass abstract control model of the USB communication device class specification.

This module supports the following functions.

- Checking of connected devices
- Implementation of communication line settings
- Acquisition of the communication line state
- Data transfer to and from a CDC device
- Connect multiple CDC devices

Limitations

HCDC is subject to the following limitations.

The structures contain members of different types. (Depending on the compiler, this may cause address misalignment of structure members.)

Terms and Abbreviations

Terms and abbreviations used in this document are listed below.

APL	:	Application program
CDC	:	Communications devices class
CDCC	:	Communications Devices Class — Communications Class Interface
CDCD	:	Communications Devices Class — Data Class Interface
HCD	:	Host control driver of USB-BASIC-FW
HCDC	:	Host communication devices class
HDCD	:	Host device class driver (device driver and USB class driver)
HUBCD	:	Hub class sample driver
MGR	:	Peripheral device state manager of HCD
Scheduler	:	Used to schedule functions, like a simplified OS.
Task	:	Processing unit
USB	:	Universal Serial Bus
USB-BASIC-FW	:	USB basic firmware for Renesas USB MCU

2. Software Configuration

Figure 2.1 shows a block diagram of HCDC, and Table 2-1 lists the modules.

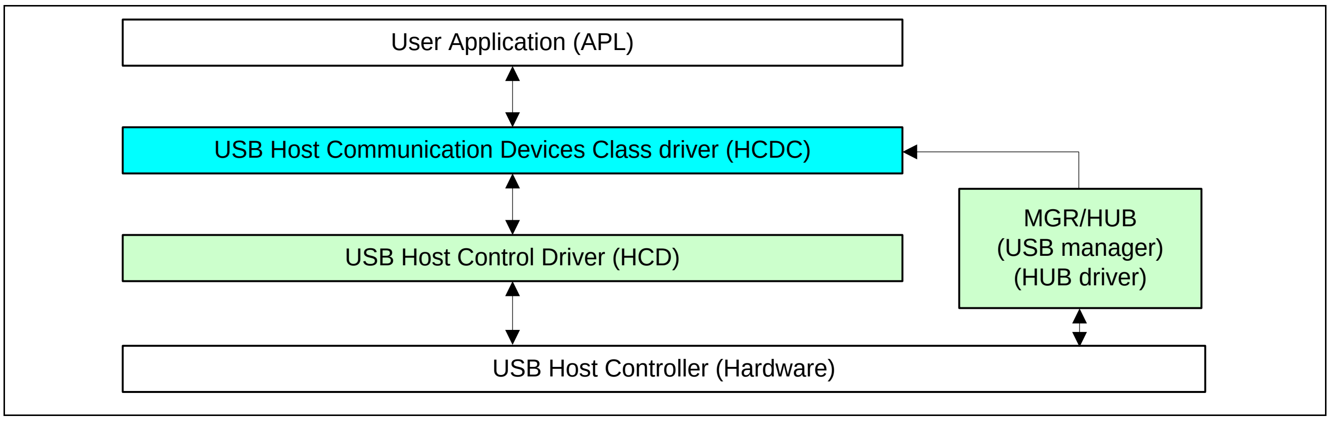

User Application (APL)
USB Host Communication Devices Class driver (HCDC)
USB Host Control Driver (HCD)
MGR/HUB
(USB manager)
(HUB driver)
USB Host Controller (Hardware)

Figure 2.1 Software Block Diagram

Table 2-1 Modules

Module Name	Description
APL	User application program. (Please prepare for your system)
HCDC	USB Host Communications Device Class Driver. CDC device access. Requests CDC requests command and the data transfer from APL to HCD.
MGR / HUB	USB Manager / HUB class driver. (USB-BASIC-F/W) Enumerates the connected devices and starts HCDC. Performs device state management.
HCD	USB host Hardware Control Driver. (USB-BASIC-F/W)

3. USB Host Communication Device Class Driver (HCDC)

This software conforms to the Abstract Control Model (ACM) subclass of the Communication Device Class specification, as specified in detail in the PSTN Subclass document listed in "Related Documents".

The Abstract Control Model subclass is a technology that bridges the gap between USB devices and earlier modems (employing RS-232C connections), enabling use of application programs designed for older modems.

3.1 Basic Functions

This software conforms to the Abstract Control Model subclass of the communication device class specification.

The main functions of HCDC are to:

1. Send class requests to the CDC peripheral
2. Transfer data to and from the CDC peripheral
3. Receive communication error information from the CDC peripheral

3.2 Abstract Control Model Class Requests - Host to Device

The HCDC supports the following ACM class requests.

Table 3-1 CDC Class Requests

Request	Code	Description
SetLineCoding	0x20	Makes communication line settings. (Communication speed, data length, parity bit, and stop bit length).
GetLineCoding	0x21	Acquires the communication line setting state.
SetControlLineState	0x22	Makes communication line control signal (RTS, DTR) settings.

For details concerning the Abstract Control Model requests, refer to Table 11, "Requests - Abstract Control Model" in "USB Communications Class Subclass Specification for PSTN Devices", Revision 1.2.

3.2.1 SetLineCoding

The SetLineCoding data format is shown Table 3-2.

Table 3-2 SetLineCoding Data Format

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0x21	SET_LINE_CODING (0x20)	0x0000	0x0000	0x0000	Line Coding Structure See Table 3-3, Line Coding Structure Format

Line Coding Structure Format is shown Table 3-3.

Table 3-3 Line Coding Structure Format

Offset	Field	Size	Value	Description
0	dwDTERate	4	Number	Data terminal speed (bps)
4	bCharFormat	1	Number	Stop bits 0 - 1 stop bit, 1 - 1.5 stop bits, 2 - 2 stop bits
5	bParityType	1	Number	Parity 0 - None, 1 - Odd, 2 - Even, 3 - Mask, 4 - Space
6	bDataBits	1	Number	Data bits (5, 6, 7, 8)

3.2.2 GetLineCoding

The GetLineCoding data format is shown Table 3-4.

Table 3-4 GetLineCoding Data Format

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0xA1	GET_LINE_CODING (0x21)	0x0000	0x0000	0x0007	Line Coding Structure See Table 3-3, Line Coding Structure Format

3.2.3 SetControlLineState

The SetControlLineState data format is shown below.

Table 3-5 SetControlLineState Data Format

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0x21	SET_CONTROL_LINE_STATE (0x22)	Control Signal Bitmap See table 3-6, Control Signal Bitmap Format	0x0000	0x0000	None

Table 3-6 Control Signal Bitmap

Bit Position	Description
D15 to D2	Reserved
D1	DCE transmit function control 0 - Deactivate carrier, 1 - Activate carrier
D0	Notification of DTE ready state 0 - Not Present, 1 - Present

3.3 ACM Notifications from Device to Host

The class notifications supported and not supported by the software are shown Table 3-7.

Table 3-7 CDC Class Notifications

Notification	Code	Description
SERIAL_STATE	0x20	Notification of serial line state

3.3.1 SerialState

The SerialState data format is shown below.

Table 3-8 SerialState Data Format

bmRequestType	bRequest	wValue	wIndex	wLength	Data
0xA1	SERIAL_STATE (0x20)	0x0000	0x0000	0x0000	UART State bitmap See table 3-9, UART State bitmap Format

Table 3-9 UART State bitmap Format

Bits	Field	Description
D15 to D7		Reserved
D6	bOverRun	Overrun error detected
D5	bParity	Parity error detected
D4	bFraming	Framing error detected
D3	bRingSignal	INCOMING signal (ring signal) detected
D2	bBreak	Break signal detected
D1	bTxCarrier	Data Set Ready: Line connected and ready for communication
D0	bRxCarrier	Data Carrier Detect: Carrier detected on line

3.4 Structures

3.4.1 HCDC Request Structure

Table 3-10 describes the “UART settings” parameter structure used for the CDC requests *SetLineCoding* and *GetLineCoding*.

Table 3-10 USB_HCDC_LineCoding_t Structure

Type	Member	Description
uint32_t	dwDTERate	Line speed (Unit : bps)
uint8_t	bCharFormat	Stop bits setting
uint8_t	bParityType	Parity setting
uint8_t	bDataBits	Data bit length

Table 3-11 describes the “UART settings” parameter structure used for the CDC requests *SetControlLineState*.

Table 3-11 USB_HCDC_ControlLineState_t Structure

Type	Member	Description
uint16_t (D1)	bRTS:1	Carrier control for half duplex modems 0 - Deactivate carrier, 1 - Activate carrier
uint16_t (D0)	bDTR:1	Indicates to DCE if DTE is present or not 0 - Not Present, 1 - Present

3.4.2 CDC Notification Format

The host is notified of the “*SerialState*” when a change in the UART port state is detected. Table 3-12 describes the structure of the UART State bitmap.

Table 3-12 USB_HCDC_SerialState_t Structure

Type	Member	Description
uint16_t (D15-D8)	rsv1:8	Reserved1
uint16_t (D7)	rsv2:1	Reserved2
uint16_t (D6)	bOverRun:1	Overrun error detected
uint16_t (D5)	bParity:1	Parity error detected
uint16_t (D4)	bFraming:1	Framing error detected
uint16_t (D3)	bRingSignal:1	Incoming signal (Ring signal) detected
uint16_t (D2)	bBreak:1	Break signal detected
uint16_t (D1)	bTxCarrier:1	Line connected and ready for communication
uint16_t (D0)	bRxCarrier:1	Carrier detected on line

3.5 Scheduler settings

Scheduler settings of HCDC is shown in Table 3-13.

Table 3-13 Scheduler settings

Function	ID	Priority	Mailbox ID	Memory Pool ID	Desctiption
R_usb_hcdc_task	USB_HCDC_TSK	USB_PRI_3	USB_HCDC_MBX	USB_HCDC_MPL	HCDC Task
R_usb_hub_task	USB_HUB_TSK	USB_PRI_3	USB_HUB_MBX	USB_HUB_MPL	HUB Task
R_usb_hstd_MgrTask	USB_MGR_TSK	USB_PRI_2	USB_MGR_MBX	USB_MGR_MPL	MGR Task
r_usb_hstd_HciTask	USB_HCI_TSK	USB_PRI_1	USB_HCI_MBX	USB_HCI_MPL	HCD Task

3.6 API

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

Please modify r_usb_hcdc_config.h when User sets the module configuration option.

Table 3-14 shows the option name and the setting value.

Table 3-14 Configuration options

Name	Default	Description
MAX_DEVICE_NUM	4	Max connect device number
INIT_COM_SPEED	USB_HCDC_SPEED_9600	SetLineCoding Setting
INIT_COM_DATA_BIT	USB_HCDC_DATA_BIT_8	
INIT_COM_STOP_BIT	USB_HCDC_STOP_BIT_1	
INIT_COM_PARITY	USB_HCDC_PARITY_BIT_NONE	

The HCDC API is shown in Table 3-15.

Table 3-15 List of HCDC API Functions

Function	Description
R_usb_hcdc_Task	HCDC task
R_usb_hcdc_driver_start	Driver task start setting for HCDC
R_usb_hcdc_class_request	Sends CDC class request
R_usb_hcdc_send_data	USB send processing
R_usb_hcdc_receive_data	USB receive processing
R_usb_hcdc_serial_state_trans	Class notification Serial State processing
R_usb_hcdc_set_line_coding	SetLineCoding request
R_usb_hcdc_get_line_coding	GetLineCoding request
R_usb_hcdc_set_control_line_state	SetControlLineState request

3.6.1 R_usb_hcdc_task

HCDC task

Format

void	R_usb_hcdc_task (void)

Argument

—	—

Return Value

—	—

Description

The HCDC task processes requests from the application, and notifies the application of the results.

Note

Call in the scheduler process of the loop.

Example

```
void usb_apl_task_switch(void)
{
  while( 1 )
  {
    /* Scheduler */
    R_usb_cstd_Scheduler();

    if( USB_FLGSET == R_usb_cstd_CheckSchedule() )
    {
        R_usb_hstd_MgrTask();    /* MGR Task */
        R_usb_hhub_Task();       /* HUB Task */
        R_usb_hcdc_task();       /* HCDC Task */
    }
  }
}
```

3.6.2 R_usb_hcdc_driver_start

HCDC driver task init

Format

void	usb_hcdc_driver_start (void)

Argument

—	—

Return Value

—	—

Description

This function set priority the HCDC driver task.

Note

Call this API from the user application at user system initialization.

Example

```
void usb_hcdc_task_start( void )
{
  hcdc_registration();            /* Host Application Registration */
  R_usb_hcdc_driver_start();      /* Host Class Driver Task Start Setting */
}
```

3.6.3 R_usb_hcdc_class_check

Check descriptor

Format

void	R_usb_hcdc_class_check (uint16_t **table)

Argument

**table	Device information table
	[0] : Device Descriptor
	[1] : Configuration Descriptor
	[2] : Interface Descriptor
	[3] : Descriptor Check Result
	[4] : HUB Classification
	[5] : Port Number
	[6] : Transmission Speed
	[7] : Device Address

Return Value

—	—

Description

This is a class driver registration function. It is registered to the driver registration structure member *classcheck*, as a callback function during HCDC registration at startup and called when a configuration descriptor is received during enumeration.

This function references the endpoint descriptor in the peripheral device configuration descriptor, then edits the pipe information table and checks the pipe information of the pipes to be used.

Note

—

Example

```
void  usb_hcdc_registration(void)
{
  USB_HCDREG_t  driver;

  driver.classcheck   = &R_usb_hcdc_class_check;

  R_usb_hstd_DriverRegistration(&driver);
}
```

3.6.4 R_usb_hcdc_send_data

Host send data

Format

USB_ER_t	R_usb_hcdc_send_data (uint16_t pipe_id uint8_t *table, uint32_t size, USB_UTR_CB_t complete)

Argument

pipe_id	Pipe ID
*table	Pointer to Transmit data buffer address
size	Transfer size
complete	Process completion notice callback function

Return Value

—	Error code (USB_OK / USB_ERROR)

Description

This function transfers the USB data in the specified transmit size from the specified address.

When the transmit processing is complete, the callback function is called.

Note

The USB transmit processing results are obtained by argument in the callback function.

Example

```
void cdc_data_transfer(uint16_t devadr)
{
  uint16_t   pipe_id;
  uint8_t    send_data[] = {0x01,0x02,0x03,0x04,0x05}; /* Data buff */
  uint32_t   size = 5;                                  /* Data size */

   pipe_id = R_usb_hstd_GetPipeID(devadr, USB_EP_BULK, USB_EP_OUT, 1);

  R_usb_hcdc_send_data(pipe_id, send_data, size, &usb_complete);
}

/* Callback function */
void  usb_complete(USB_UTR_t *utr);
{
  /* Describe the processing performed when the USB transmit is completed. */
}
```

3.6.5 R_usb_hcdc_receive_data

Host receive data

Format

```
USB_ER_t          R_usb_hcdc_receive_data (uint16_t pipe_id
                                        uint8_t *table,
                                        uint32_t size,
                                        USB_CB_t complete )
```

Argument

pipe_id	Pipe ID
*table	Pointer to transmit data buffer address
size	Transfer size
complete	Process completion notice callback function

Return Value

—	Error code (USB_OK / USB_ERROR).

Description

This function requests USB data reception from the USB driver (HCD).

When data reception ends (specified data size reached, short packet received, error occurred), the callback function is called. Information on remaining receive data (length, status, error count and transfer end) is determined by the parameters of the callback.

USB receive data is stored in the area given by the specified address.

Note

The USB transmit process results are obtained from the argument in the callback function.

Example

```
void cdc_data_transfer(uint16_t devadr)
{
  uint16_t   pipe_id;
  uint8_t    receive_data[64];                    /* Data buff */
  uint32_t   size = 64;                           /* Data size */

  pipe_id = R_usb_hstd_GetPipeID(devadr, USB_EP_BULK, USB_EP_IN, 1);

  R_usb_hcdc_receive_data(pipe_id, receive_data, size, &usb_complete);
}

/* Callback function */
void  usb_complete(USB_UTR_t *utr)
{
  /* Describe the processing performed when the USB receive is completed. */
}
```

3.6.6 R_usb_hcdc_serial_state_trans

Handle CDC class and serial state info from peripheral

Format

USB_ER_t	R_usb_hcdc_serial_state_trans (uint16_t pipe_id, uint8_t *table, USB_UTR_CB_t complete)

Argument

pipe_id	Pipe ID
*table	Pointer to transmit data buffer address
complete	Process completion notice callback function

Return Value

—	Error code (USB_OK / USB_ERROR).

Description

This function receives the CDC class notification (SerialState) from the peripheral device.

Callback function is called after the completion of reception.

The serial status is received when the callback function is triggered.

Note

1. Transfer data area has to allocate more than 10 bytes.
2. For information concerning the serial status bit pattern, refer to "Table 3-9".
3. The USB transmit results are obtained from the argument in the call-back function.

Example

```
void cdc_data_transfer(uint16_t devadr)
{
   uint16_t   pipe_id;
  uint8_t    serial_data[10];              /* Data buff */

   pipe_id = R_usb_hstd_GetPipeID(devadr, USB_EP_INT, USB_EP_IN, 0);

  R_usb_hcdc_serial_state_trans(pipe_id, serial_data, &usb_complete);
}


/* Callback function */
void usb_complete(USB_UTR_t *utr)
{
  uint16_t *status;

  status = (uint16_t *)utr->tranadr;    /* Status set */
  /* [0] bmRequestType/bRequest */
  /* [1] wValue */
  /* [2] wIndex */
  /* [3] wLength */
  /* [4] data : Serial State(UART State bitmap) */
  check_status(status[4]);
}
```

3.6.7 R_usb_hcdc_set_line_coding

SetLineCoding request

Format

USB_ER_t	R_usb_hcdc_set_line_coding (uint16_t devadr, USB_HCDC_LineCoding_t *p_linecoding, USB_UTR_CB_t complete)

Argument

devadr	Device address
*p_linecoding	Parameter of UART setting
complete	Process completion notice callback function

Return Value

—	Error code (USB_OK / USB_ERROR).

Description

This API function the SetLineCoding request processing.

Note

The USB transmit process results are obtained from the argument in the callback function.

Example

```
 void cdc_class_request(uint16_t devadr)
{
   USB_HCDC_LineCoding_t     cdc_line_coding;

   cdc_line_coding.dwDTERate    = USB_HCDC_SPEED_9600;
   cdc_line_coding.bDataBits    = USB_HCDC_DATA_BIT_8;
   cdc_line_coding.bCharFormat  = USB_HCDC_STOP_BIT_1;
   cdc_line_coding.bParityType  = USB_HCDC_PARITY_BIT_NONE;

   R_usb_hcdc_set_line_coding(devadr, &cdc_line_coding, &cdc_setlinecoding_cb);
 }

 /* Callback function */
 void cdc_setlinecoding_cb(USB_UTR_t *utr)
 {
   /* Describe the processing performed when the USB receive is completed. */
 }
```

3.6.8 R_usb_hcdc_get_line_coding

GetLineCoding request

Format

```
USB_ER_t          R_usb_hcdc_get_line_coding (uint16_t devadr,
                                      USB_HCDC_LineCoding_t *p_linecoding,
                                      USB_UTR_CB_t complete )
```

Argument

devadr	Device address
*p_linecoding	Parameter of UART setting
complete	Process completion notice callback function

Return Value

—	Error code (USB_OK / USB_ERROR).

Description

This API function the GetLineCoding request processing.

Note

The USB transmit process results are obtained from the argument in the callback function.

Example

```
 void cdc_class_request(uint16_t devadr)
{
   USB_HCDC_LineCoding_t     cdc_line_coding;

   R_usb_hcdc_get_line_coding(devadr, &cdc_line_coding, &cdc_getlinecoding_cb);
 }

 /* Callback function */
 void cdc_getlinecoding_cb(USB_UTR_t *utr)
 {
   /* Describe the processing performed when the USB receive is completed. */
 }
```

3.6.9 R_usb_hcdc_set_control_line_state

SetControlLineState request

Format

```
USB_ER_t        R_usb_hcdc_set_control_line_state (uint16_t devadr,
                                                   uint16_t dtr,
                                                   uint16_t rts,
                                                   USB_UTR_CB_t complete )
```

Argument

devadr	Device address
dtr	RS232 signal DTR
rts	RS232 signal RTS
complete	Process completion notice callback function

Return Value

—	Error code (USB_OK / USB_ERROR).

Description

This API function the SetControlLineState request processing.

Note

The USB transmit process results are obtained from the argument in the callback function.

Example

```
void cdc_class_request(uint16_t devadr)
{
   R_usb_hcdc_set_control_line_state(devadr, USB_TRUE, USB_TRUE, &cdc_complete);
}

/* Callback function */
void cdc_complete(USB_UTR_t *utr)
{
   /* Describe the processing performed when the USB receive is completed. */
}
```

4. Sample Application

This section describes the initial settings necessary for using the USB HCDC and USB-BASIC-F/W in combination as a USB driver and presents an example of data transfer by means of processing by the main routine and the use of API functions.

4.1 Setup

Figure 4-1 shows an example operating environment for the HCDC.

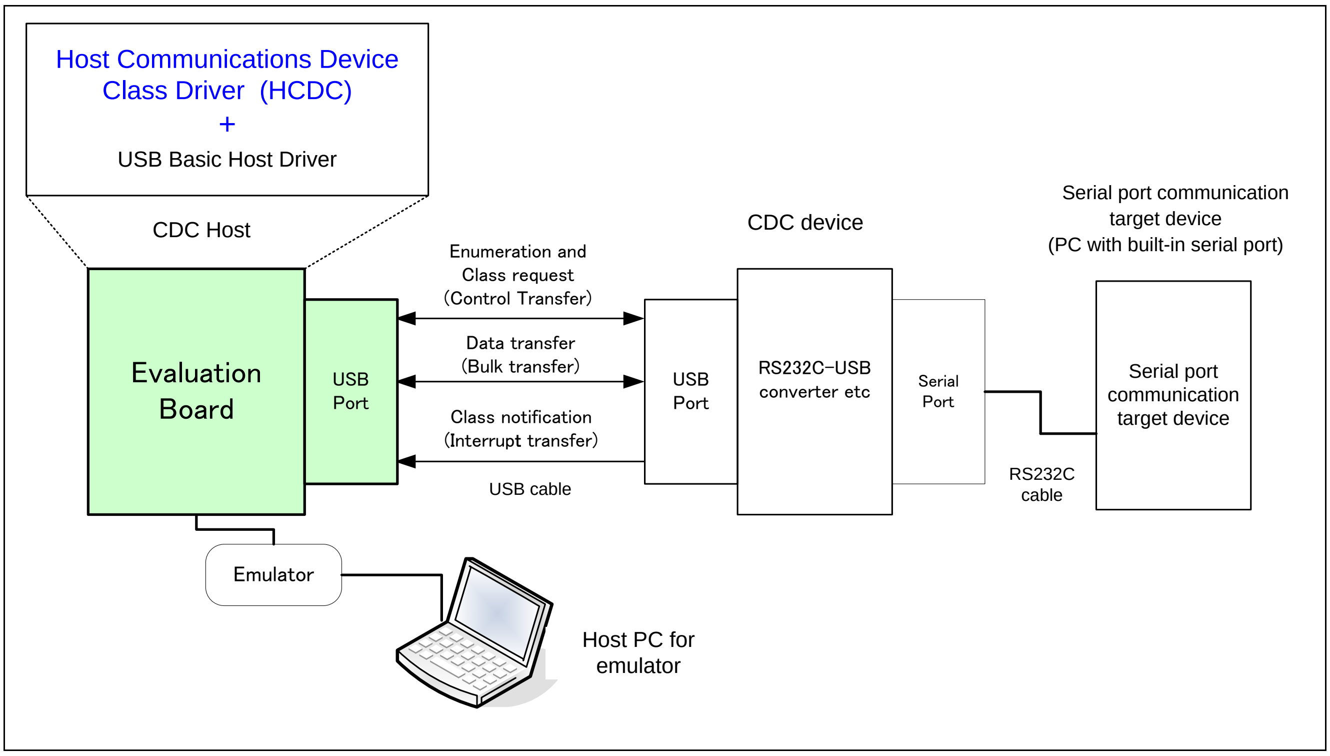

Host Communications Device Class Driver (HCDC)
+
USB Basic Host Driver
CDC Host
Evaluation Board
USB Port
Enumeration and Class request (Control Transfer)
Data transfer (Bulk transfer)
Class notification (Interrupt transfer)
USB cable
CDC device
USB Port
RS232C-USB converter etc
Serial Port
RS232C cable
Serial port communication target device (PC with built-in serial port)
Serial port communication target device
Emulator
Host PC for emulator

Figure 4-1 Example Operating Environment

4.2 Application Specifications

The main functions of HCDC sample application are as follows:

1. Sends receive (Bulk In transfer) requests to the CDC device and receives data.
2. Transfers received data to the CDC device by means of Bulk Out transfers (loopback).
3. Set RTS and DTR by the class request SET_CONTROL_LINE_STATE.
4. Set the communication speed, number of data bits, number of stop bits, the parity bit, by the class request SET_LINE_CODING.
5. Acquires the communication setting values of the CDC device by the class request GET_LINE_CODING.
6. Reports changes in the line status to the application program.

4.3 Data Transfer Image

Figure 4-2 shows the data transfer image.

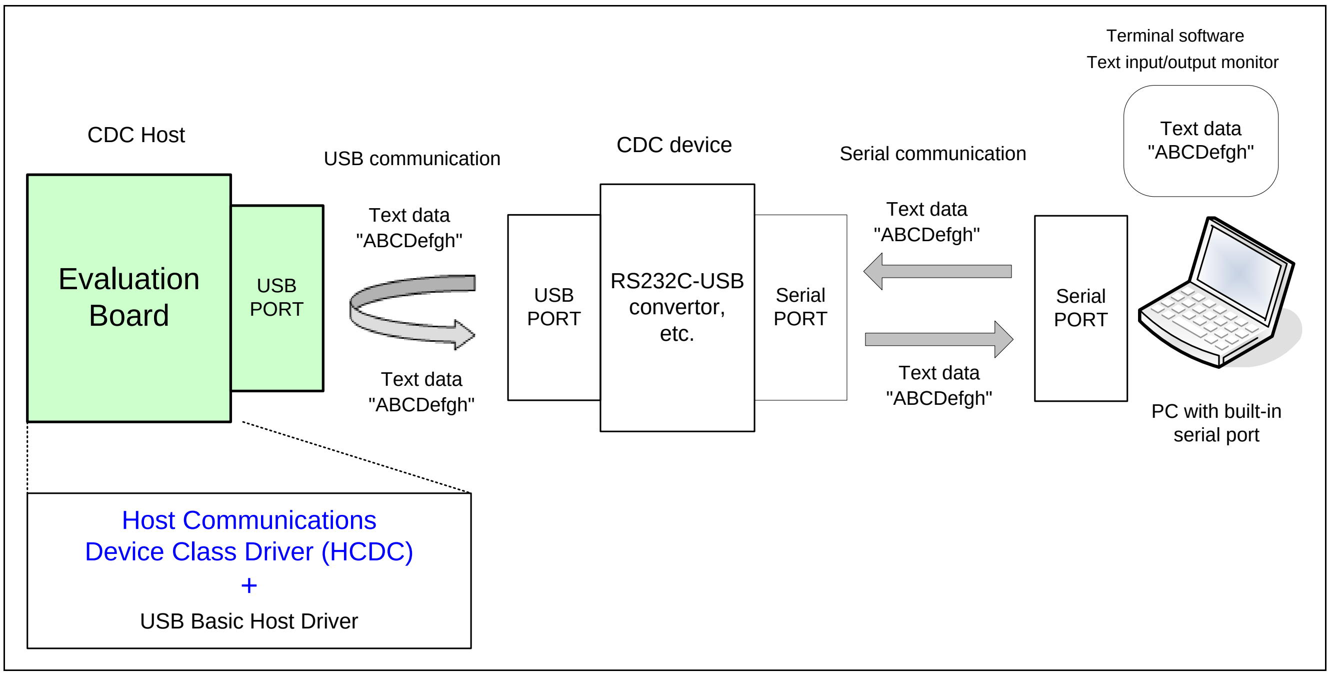


Figure 4-2 Data Transfer (Loopback) Image

4.4 Initial Settings of USB Driver

Sample settings are shown below.

```
void usb_hcdc_apl(void)
{
   /* MCU Pin Setting (Refer to “4.4.1”) */
   usb_mcu_setting();

   /* USB Driver Setting (Refer to “4.4.2”) */
   R_usb_hstd_MgrOpen();
   R_usb_cstd_SetTaskPri(USB_HUB_TSK, USB_PRI_3);   // Note
   R_usb_hhub_Registration(USB_NULL);               // Note
   cdc_registration();
   R_usb_hcdc_driver_start();

   /* Main routine (Refer to “4.5”) */
   usb_hapl_mainloop();
}
```

[Note]

It is only necessary to call this function when the HUB will be used.

4.4.1 MCU Settings

Set the USB module according to the initial setting sequence of the hardware manual, the USB interrupt handler registration and USB interrupt enable setting.

4.4.2 USB Driver Settings

The USB driver settings consist of registering a task with the scheduler and registering class driver information for the USB-BASIC-F/W. The procedure is described below.

1. Call the USB-BASIC-F/W's API function (R_usb_hstd_MgrOpen() to register the MGR task and the HCD task with the scheduler.
2. Call the class driver API function (R_usb_hhub_Registration()) to register the HUB task with the scheduler.
3. After specifying the necessary information in the members of the class driver registration structure (USB_HCDREG_t), call the USB-BASIC-F/W's API function (R_usb_hstd_DriverRegistration() to register the class driver information.
4. Call the class driver HCDC's API function (R_usb_hcdc_driver_start()) to register the HCDC task with the scheduler.

A sample of information specified in the structure declared by USB_HCDREG_t is shown below.

```
void usb_hapl_registration(void)
{
   /* Structure for the class driver registration */
   USB_HCDREG_t  driver;

   /* Class Code which is defined in the USB specification setting*/
   driver.ifclass     = (uint16_t)USB_IFCLS_CDCC;
   /* TPL setting */
   driver.tpl         = (uint16_t*)&usb_gapl_devicetpl; // Note 1
   /* Set the class check function which is called in the enumeration. */
   driver.classcheck  = &R_usb_hcdc_class_check;
   /* Set the function which is called when completing the enumeration */
   driver.devconfig   = &cdc_configured;
   /* Set the function which is called when disconnecting USB device */
   driver.devdetach   = &cdc_detach;
   /* Set the function which is called when changing the suspend state */
   driver.devsuspend  = &cdc_suspend;
   /* Set the function which is called when resuming from the suspend state */
   driver.devresume   = &cdc_resume;

   /* Register the class driver information to HCD */
   R_usb_hstd_DriverRegistration(&driver);
}
```

[Note]

1. TPL(Target Peripheral List) need to be defined in the application program. Refer to USB Basic Firmware application note (Document No.R01AN2633EJ) about TPL.

4.5 Processing by Main Routine

After the USB driver initial settings, call the scheduler (R_usb_cstd_Scheduler()) from the main routine of the application. Calling R_usb_cstd_Scheduler() from the main routine causes a check for events. If there is an event, a flag is set to inform the scheduler that an event has occurred. After calling R_usb_cstd_Scheduler(), call R_usb_cstd_CheckSchedule() to check for events. Also, it is necessary to run processing at regular intervals to get events and perform the appropriate processing.*[1]

```
void usb_hapl_mainloop(void)
{
  while(1)  // Main routine
  {
    // Confirming the event and getting (Note1)
    R_usb_cstd_Scheduler();

    // Judgment whether the event is or not
    if(USB_FLGSET == R_usb_cstd_CheckSchedule())
    {                                                                  (Note 2)
        R_usb_hstd_MgrTask();   // MGR task
        R_usb_hhub_Task();      // HUB task (Note 3)
        R_usb_hcdc_task();      // CDC task
    }
    hcdc_application();         // User application program(APL)
  }
}
```

[Note]

1. If, after getting an event with R_usb_cstd_Scheduler() and before running the corresponding processing, R_usb_cstd_Scheduler() is called again and gets another event, the first event is discarded. After getting an event, always call the corresponding task to perform the appropriate processing.
2. Be sure to describe these processes in the main loop for the application program.
3. It is only necessary to call this function when the HUB will be used.

4.5.1 APL

The application comprises two parts: initial settings and main loop. An overview of the processing in these two parts is provided below.

1. The APL manages the states and the events associated with them. The APL first checks the state of the connected device (see Table 4-1). This state is stored in a member of a structure managed by the APL.(see 4.5.2)
2. Next, the APL checks the events related to the state (see Table 4-2) and performs the associated processing. After processing an event, the APL changes the state if necessary. These events are stored in members of a structure managed by the APL. (see 4.5.2)

An overview of the processing performed by the APL is shown below:

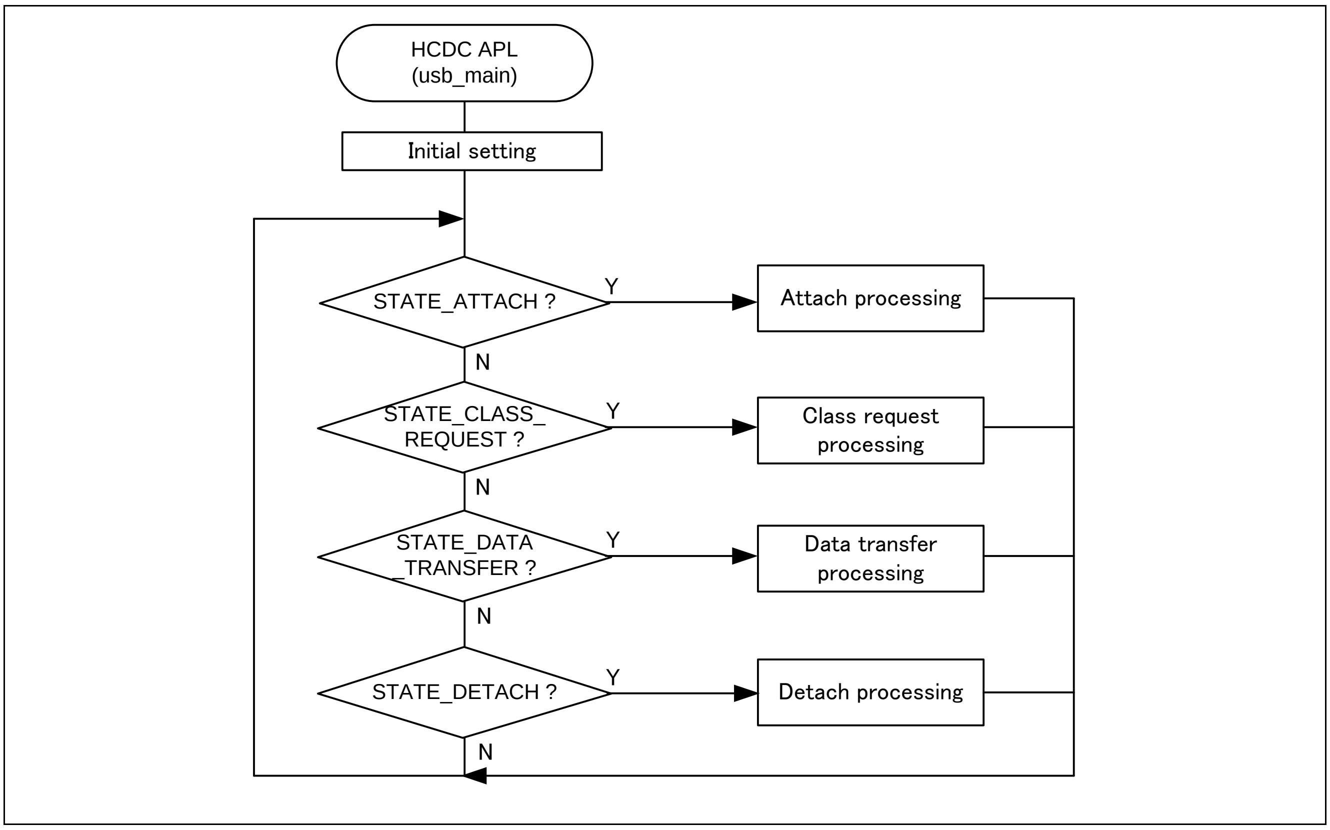


Figure 4-3 Main Loop processing

4.5.2 State and Event Management

Members of the following structure are used to manage states and events. This structure is prepared by the APL.

```
typedef struct DEV_INFO             /* Structure for CDC device control */
{
    uint16_t    state;              /* State for application */
    uint16_t    event_cnt;          /* Event count */
    uint16_t    event[EVENT_MAX];   /* Event no. */
    uint16_t    in_pipe;            /* Use pipe no. */
    uint16_t    out_pipe;           /* Use pipe no. */
    uint16_t    status_pipe;        /* Use pipe no. */
    uint16_t    cr_seq;             /* Class Request Sequence */
    uint16_t    trans_len;          /* TX Length */
    uint8_t     trans_data[ CDC_DATA_LEN + 4 ];     /* RX Data */
    uint8_t     serial_state_data[USB_HCDC_SERIAL_STATE_MSG_LEN];
    USB_HCDC_SerialState_t  serial_state_bitmap;
    USB_HCDC_LineCoding_t   com_parm;    /* Set Line Coding parameter */
} DEV_INFO_t;
```

Table 4-1 List of States

State	State Processing Overview	Related Event
STATE_ATTACH	Attach processing	EVENT_CONFIGURD
STATE_CLASS_REQUEST	Class request processing	EVENT_CLASS_REQUEST_START
		EVENT_CLASS_REQUEST_COMPLETE
STATE_DATA_TRANSFER	Data transfer processing	EVENT_USB_READ_START
		EVENT_USB_READ_COMPLETE
		EVENT_USB_WRITE_START
		EVENT_USB_WRITE_COMPLETE
		EVENT_NOTIFY_READ_START
		EVENT_NOTIFY_READ_COMPLETE

Table 4-2 List of Events

Event	Outline
EVENT_CONFIGURD	USB device connecting completion
EVENT_CLASS_REQUEST_START	Request of sending the class request
EVENT_CLASS_REQUEST_COMPLETE	Class request complete
EVENT_USB_READ_START	Data read request
EVENT_USB_READ_COMPLETE	Data read complete
EVENT_USB_WRITE_START	Data write request
EVENT_USB_WRITE_COMPLETE	Data write complete
EVENT_COM_NOTIFY_RD_START	Notification receive request
EVENT_COM_NOTIFY_RD_COMPLETE	Notification receive complete
EVENT_DETACH	Detach
EVENT_NONE	No event

An overview of the processing associated with each state is provided below.

1. Attach Processing (STATE_ATTACH)

== Outline ==

In this state, processing is performed to notify that a CDC device has attached and that enumeration has finished, and the state changes to STATE_CLASS_REQUEST.

== Description ==

① In the APL, first the initialization function sets the state to STATE_ATTACH and the event to EVENT_NONE.

② The state continues to be STATE_ATTACH until an CDC device is connected, and *cdc_connect_wait()* is called.

③ When a CDC device is connected and enumeration completes, the callback function *cdc_configured()* is called by the USB driver, this callback function issues the event EVENT_CONFIGURD. The callback function *cdc_configured()* is specified in the member *devconfig* of structure *USB_HCDREG_t*.

④ In event EVENT_CONFIGURD, the state changes to STATE_CLASS_REQUEST and the event EVENT_CLASS_REQUEST_START is issued.

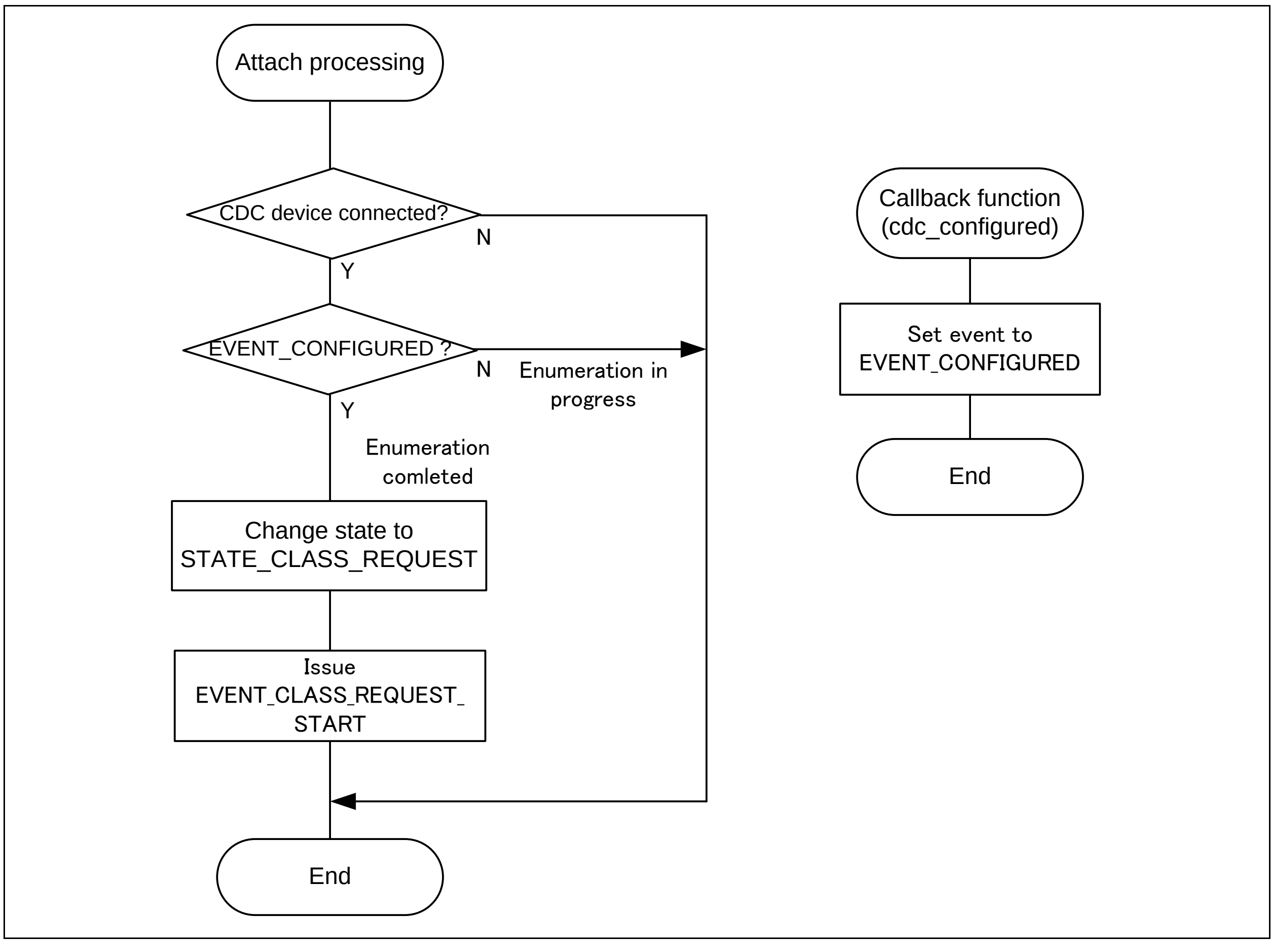

Attach processing
CDC device connected?
N
Y
EVENT_CONFIGURED ?
N
Enumeration in progress
Y
Enumeration comleted
Change state to STATE_CLASS_REQUEST
Issue EVENT_CLASS_REQUEST_START
End
Callback function (cdc_configured)
Set event to EVENT_CONFIGURED
End

Figure 4-4 Flowchart of Attach Processing

2. Class Request Processing (STATE_CLASS_REQUEST)

== Outline ==

In this state, processing is performed to transmit class requests to the CDC device. When transmission of a separately specified class requests finishes, the state changes to STATE_DATA_TRANSFER.

== Description ==

① In this state, first EVENT_CLASS_REQUEST_START is processed, and a class request transmit request is sent to the USB driver.

② When class request transmit processing completes, the callback function *cdc_class_request_complete()* is called. This callback function issues EVENT_CLASS_REQUEST_COMPLETE.

③ The processing described in ① and ② is repeated, and the class requests SetControlLineState, SetLineCoding and GetLineCoding are transmitted to the CDC device in sequence.

④ When transmission of the class request GetLineCoding finishes, the state changes to STATE_DATA_TRANSFER, and the event is set to EVENT_USB_READ_START.

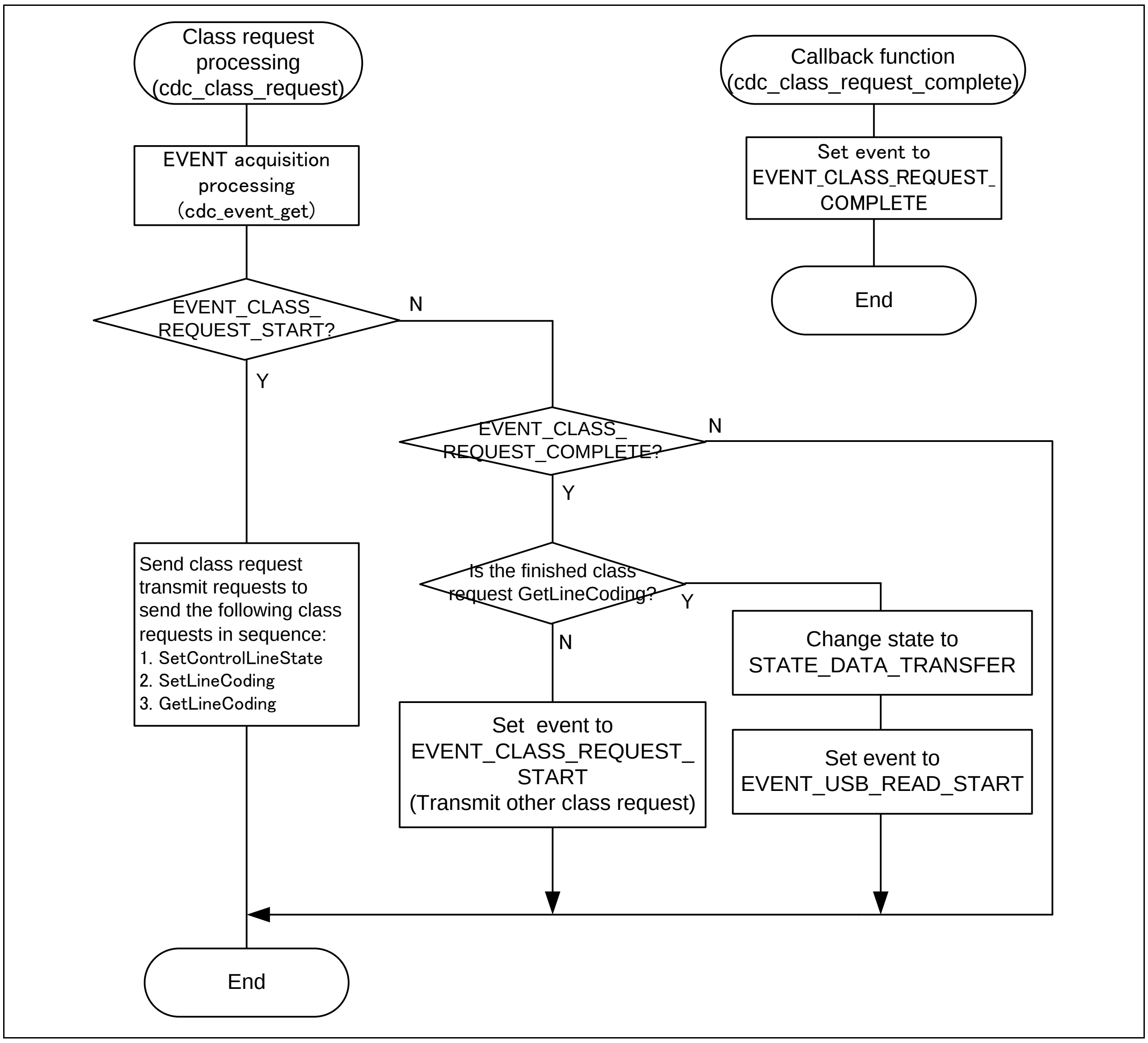

Class request processing (cdc_class_request)
EVENT acquisition processing (cdc_event_get)
EVENT_CLASS_ REQUEST_START?
N
Y
EVENT_CLASS_ REQUEST_COMPLETE?
N
Y
Is the finished class request GetLineCoding?
Y
N
Send class request transmit requests to send the following class requests in sequence:
1. SetControlLineState
2. SetLineCoding
3. GetLineCoding
Set event to EVENT_CLASS_REQUEST_ START (Transmit other class request)
Change state to STATE_DATA_TRANSFER
Set event to EVENT_USB_READ_START
End
Callback function (cdc_class_request_complete)
Set event to EVENT_CLASS_REQUEST_ COMPLETE
End

Figure 4-5 Flowchart of Class Request Processing

3. Data Transfer Processing (STATE_DATA_TRANSFER)

== Outline ==

This loopback processing routine receives data from the CDC device and then transmits the same data, unmodified, back to the CDC device.

== Description ==

① When transmission of the class request to the CDC device completes, the state transitions to STATE_DATA_TRANSFER. In this state, EVENT_USB_READ_START is processed and a data transfer processing request is sent to the USB driver.

② When data read processing completes, the callback function *cdc_read_complete()* is called. This callback function issues EVENT_USB_READ_COMPLETE.

③ In EVENT_USB_READ_COMPLETE, EVENT_USB_WRITE_START is set to the event.

④ In EVENT_USB_WRITE_START, a data write request is sent to the USB driver in order to transmit the data received in ① above to the CDC device.

⑤ When the data write finishes, the callback function *cdc_write_complete()* is called. This callback function issues the event EVENT_USB_WRITE_COMPLETE.

⑥ In EVENT_USB_WRITE_COMPLETE the event EVENT_USB_READ_START is issued, and in the next loop ① is processed again and steps ① to ⑥ are repeated.

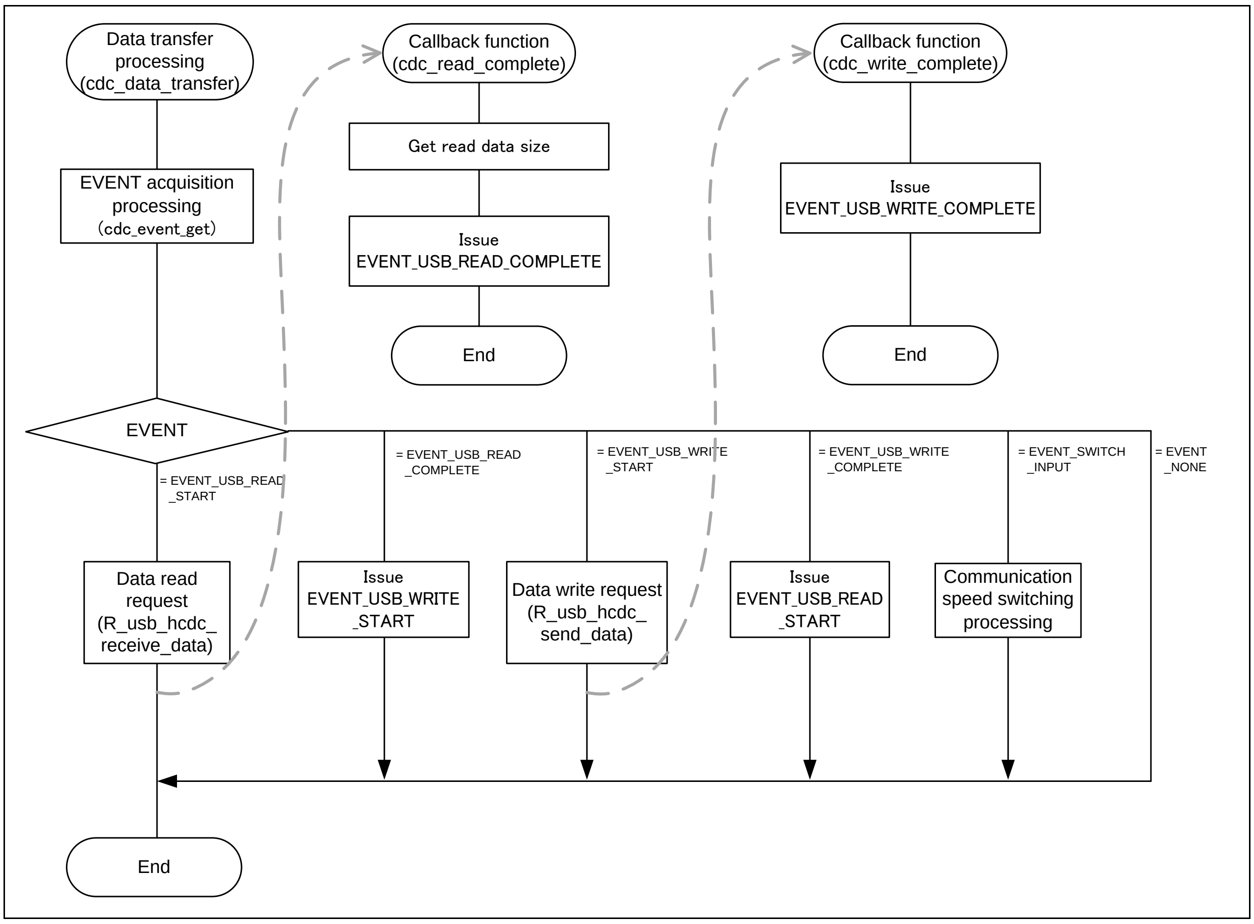

Data transfer processing (cdc_data_transfer)
Callback function (cdc_read_complete)
Callback function (cdc_write_complete)
Get read data size
EVENT acquisition processing (cdc_event_get)
Issue EVENT_USB_READ_COMPLETE
Issue EVENT_USB_WRITE_COMPLETE
End
End
EVENT
= EVENT_USB_READ _START
= EVENT_USB_READ _COMPLETE
= EVENT_USB_WRITE _START
= EVENT_USB_WRITE _COMPLETE
= EVENT_SWITCH _INPUT
= EVENT _NONE
Data read request (R_usb_hcdc_ receive_data)
Issue EVENT_USB_WRITE _START
Data write request (R_usb_hcdc_ send_data)
Issue EVENT_USB_READ _START
Communication speed switching processing
End

Figure 4-6 Flowchart of Data Transfer Processing

4. Detach Processing (STATE_DETACH)

When the connected CDC device is disconnected, the USB driver calls the callback function *cdc_detach()*. This callback function changes the state to STATE_DETACH. In STATE_DETACH, processing is performed to clear variables and change the state to STATE_ATTACH, among other things. The callback function *cdc_detach()* is the function set in the member devdetach of the structure *USB_HCDREG_t*.

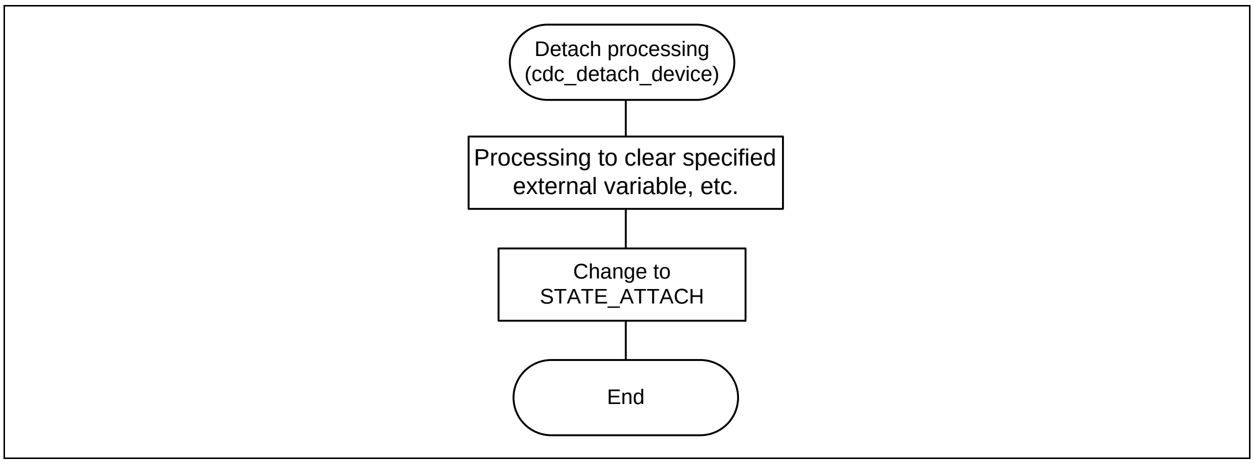

Detach processing (cdc_detach_device)
Processing to clear specified external variable, etc.
Change to STATE_ATTACH
End

Figure 4-7 Flowchart of Detach Processing

Appendix A. Changes of initial setting

USB-BASIC-F/W has been changed to "RZ/T1 group initial setting Rev.1.30".
Sample program supports IAR embedded workbench for ARM (EWARM) , DS-5 and e^2 studio.
This chapter describes the changes.

Folders and files

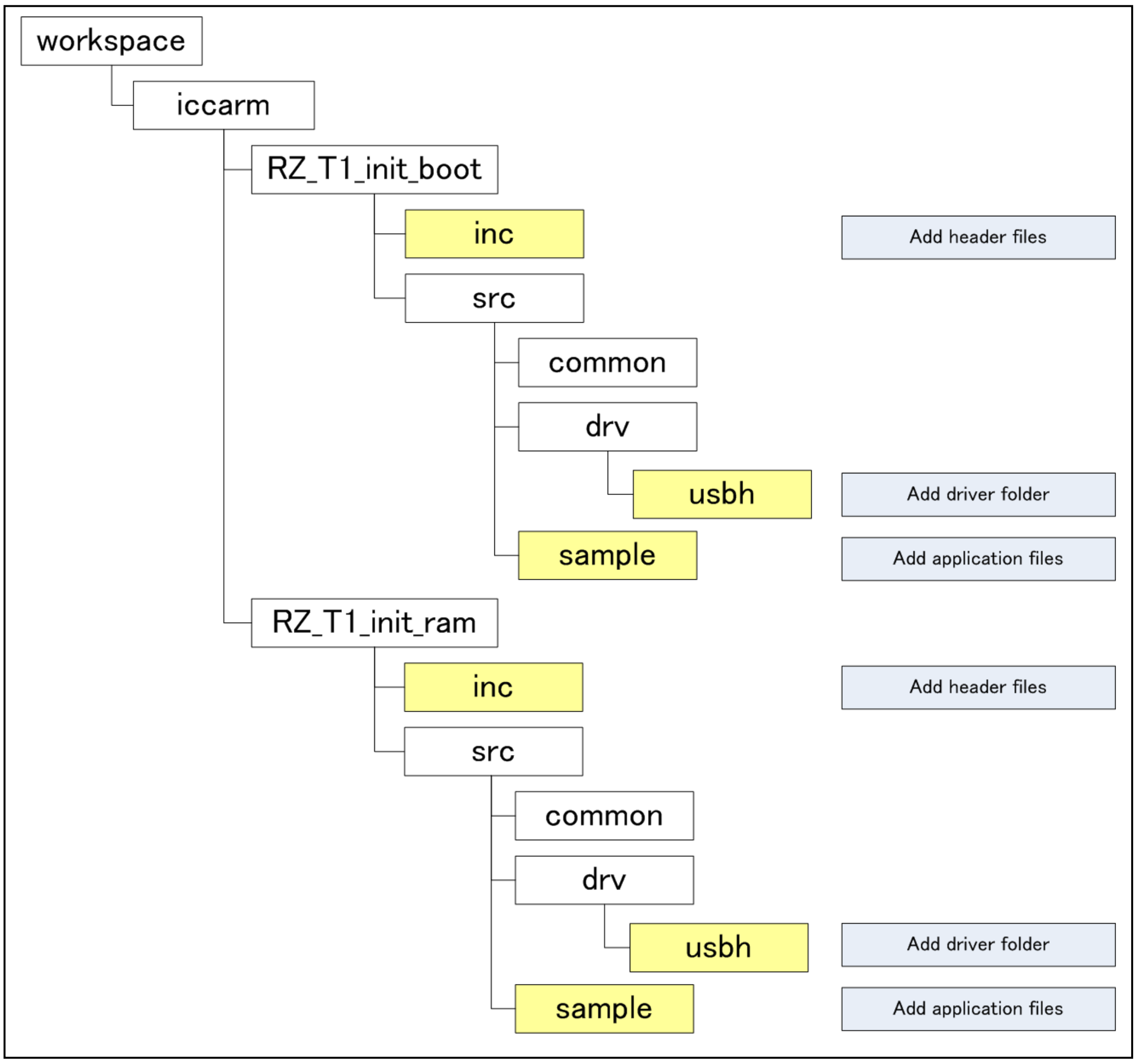
In the "RZ/T1 group initial setting Rev.1.30", different folder structure by the development environment and the boot method. Changes to each folder of all of the development environment and the boot method it is shown below.

- Add the following files in the “inc” folder.
  r_usb_basic_config.h
  r_usb_basic_if.h
  r_usb_hcdc_config.h
  r_usb_hcdc_if.h

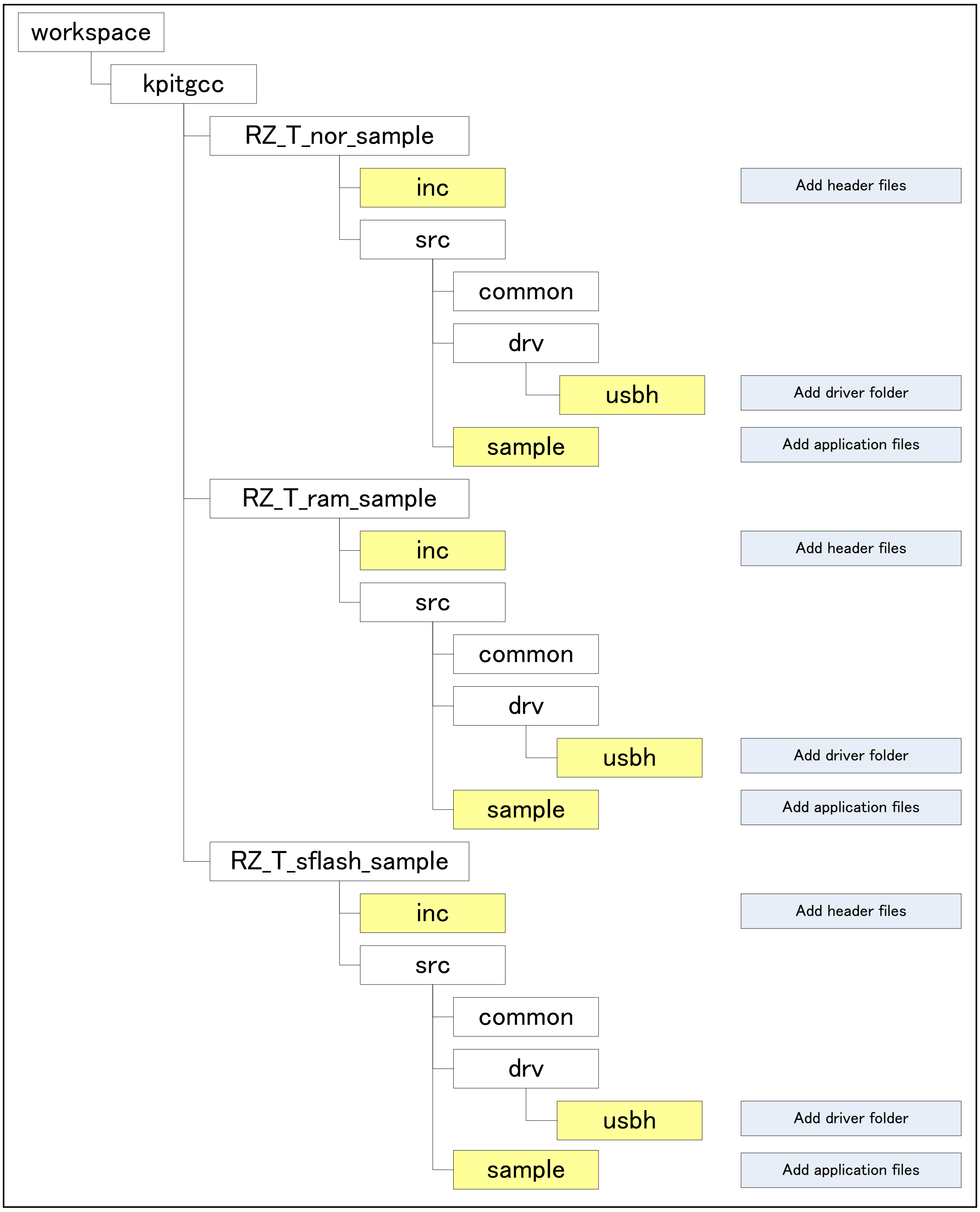
- Add the following files in the “sample” folder.
  r_usb_main.c
  r_usb_hcdc_apl.c
  r_usb_hcdc_apl.h

- Add the “usbh” folder and the following files “usbh” folder in the “drv” folder.

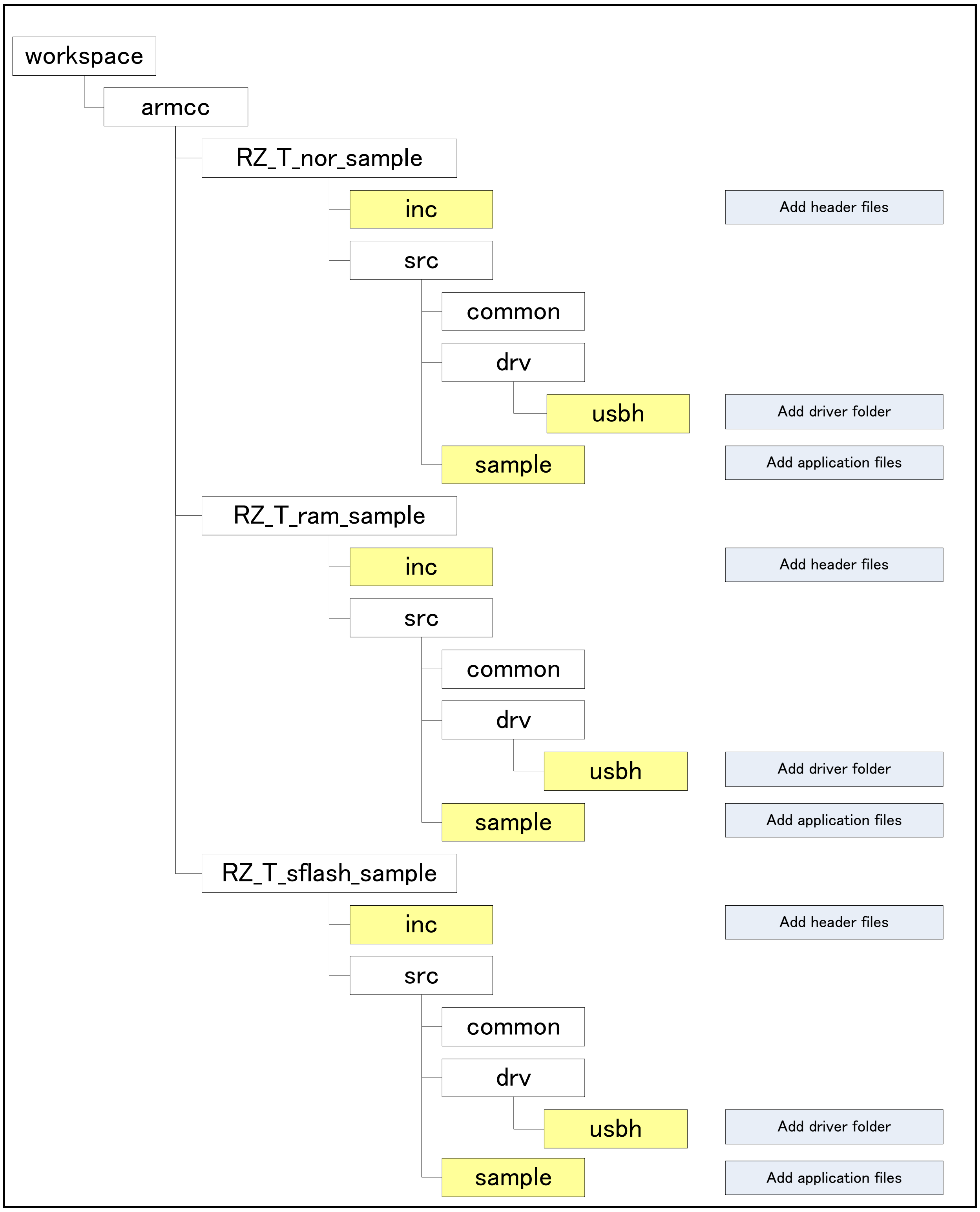
The following is the folder structure of EWARM.



The following is the folder structure of e^2 studio.



The following is the folder structure of DS-5.



Section

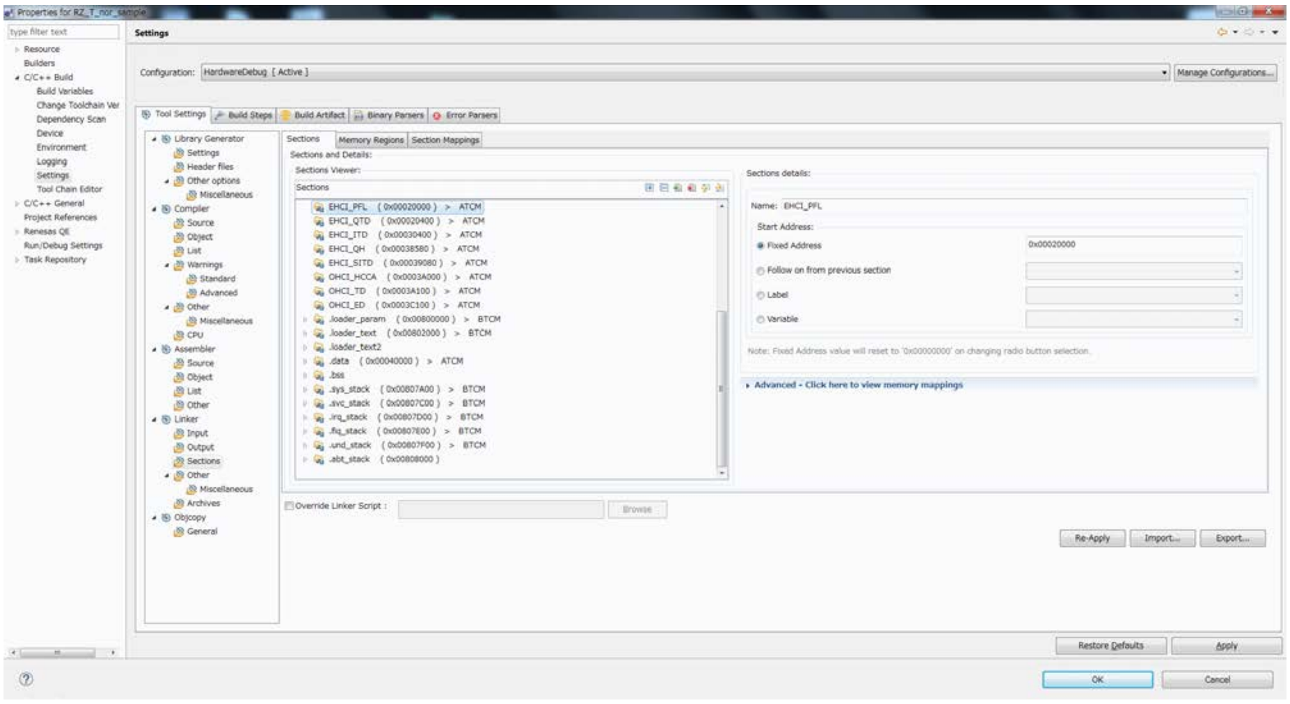
Modify the section size of the code area and a data area, and add the following section.

Section name	Address	variable	file
EHCI_PFL	0x00020000	ehci_PeriodicFrameList	r_usb_hEhciMemory.c
EHCI_QTD	0x00020400	ehci_Qtd	
EHCI_ITD	0x00030400	ehci_Itd	
EHCI_QH	0x00038580	ehci_Qh	
EHCI_SITD	0x00039080	ehci_Sitd	
OHCI_HCCA	0x0003A000	ohci_hcca	r_usb_hOhciMemory.c
OHCI_TD	0x0003A100	ohci_TdMemory	
OHCI_ED	0x0003c100	ohci_EdMemory	

e^2 studio

e^2 studio sets the section in the configuration screen.
Changes are as follows:
・Fixed address of ".data" section from 0x0007F000 to 0x00040000
・Add section setting of EHCI and OHCI.

Refer to [Project] → [Properties] → [C/C++ Build] → [Settings] → [Sections].



Sections and Details:

Sections Viewer:

```
Sections
EHCI_PFL   ( 0x00020000 )  >  ATCM
EHCI_QTD   ( 0x00020400 )  >  ATCM
EHCI_ITD   ( 0x00030400 )  >  ATCM
EHCI_QH   ( 0x00038580 )  >  ATCM
EHCI_SITD   ( 0x00039080 )  >  ATCM
OHCI_HCCA   ( 0x0003A000 )  >  ATCM
OHCI_TD   ( 0x0003A100 )  >  ATCM
OHCI_ED   ( 0x0003C100 )  >  ATCM
.loader_param   ( 0x00800000 )  >  BTCM
.loader_text   ( 0x00802000 )  >  BTCM
.loader_text2
.data   ( 0x00040000 )  >  ATCM
.bss
.sys_stack   ( 0x00807A00 )  >  BTCM
.svc_stack   ( 0x00807C00 )  >  BTCM
.irq_stack   ( 0x00807D00 )  >  BTCM
.fiq_stack   ( 0x00807E00 )  >  BTCM
.und_stack   ( 0x00807F00 )  >  BTCM
.abt_stack   ( 0x00808000 )
```

Sections details:

Name: EHCI_PFL

Start Address:
[✓ Fixed Address] 0x00020000
Follow on from previous section
Label
Variable

Note: Fixed Address value will reset to '0x00000000' on changing radio button selection.

Advanced - Click here to view memory mappings

Variable definitions in the code are as follows.

r_usb_hEhciMemory.c

```
#ifdef __GNUC__
static uint32_t         ehci_PeriodicFrameList[ USB_EHCI_PFL_SIZE ]
                __attribute__ ((section ("EHCI_PFL")));
static USB_EHCI_QH      ehci_Qh[ USB_EHCI_NUM_QH ]
                __attribute__ ((section ("EHCI_QH")));
static USB_EHCI_QTD     ehci_Qtd[ USB_EHCI_NUM_QTD ]
                __attribute__ ((section ("EHCI_QTD")));
static USB_EHCI_ITD     ehci_Itd[ USB_EHCI_NUM_ITD ]
                __attribute__ ((section ("EHCI_ITD")));
static USB_EHCI_SITD    ehci_Sitd[ USB_EHCI_NUM_SITD ]
                __attribute__ ((section ("EHCI_SITD")));
#endif  /* __GNUC__ */
```

r_usb_hOhciMemory.c

```
#ifdef __GNUC__
static USB_OHCI_HCCA_BLOCK              ohci_hcca
                __attribute__ ((section ("OHCI_HCCA")));
static USB_OHCI_HCD_TRANSFER_DESCRIPTOR ohci_TdMemory[USB_OHCI_NUM_TD]
                __attribute__ ((section ("OHCI_TD")));
static USB_OHCI_HCD_ENDPOINT_DESCRIPTOR ohci_EdMemory[USB_OHCI_NUM_ED]
                __attribute__ ((section ("OHCI_ED")));
#endif  /* __GNUC__ */
```

EWARM

EWARM sets the section in the linker setting file (.icf file).

Changes are as follows:

・Start address of RAM region from 0x00070000 to 0x00040000.
・End address of USER_PRG region from 0x0006FFFF to 0x0001FFFF.

```
define symbol __ICFEDIT_region_RAM_start__    = 0x00040000;

define symbol __region_USER_PRG_end__         = 0x0001FFFF;
```

・To the EHCI and OHCI to fixed address, adds memory region definition.

```
define region EHCI_MEM1_region = mem:[from 0x00020000 to 0x000203FF];
define region EHCI_MEM2_region = mem:[from 0x00020400 to 0x00039FFF];

define region OHCI_MEM1_region = mem:[from 0x0003A000 to 0x0003A0FF];
define region OHCI_MEM2_region = mem:[from 0x0003A100 to 0x0003FFFF];

do not initialize  { section EHCI_PFL, section EHCI_QH, section EHCI_QTD, section EHCI_ITD, section
EHCI_SITD, section OHCI_HCCA, section OHCI_TD, section OHCI_ED };

place in EHCI_MEM1_region { section EHCI_PFL };
place in EHCI_MEM2_region { section EHCI_QH, section EHCI_QTD, section EHCI_ITD, section EHCI_SITD };

place in OHCI_MEM1_region { section OHCI_HCCA };
place in OHCI_MEM2_region { section OHCI_TD, section OHCI_ED };
```

Variable definitions in the code are as follows.

r_usb_hEhciMemory.c

```
#ifdef __ICCARM__
#pragma location="EHCI_PFL"
static uint32_t                 ehci_PeriodicFrameList[ USB_EHCI_PFL_SIZE ];
#pragma location="EHCI_QH"
static USB_EHCI_QH              ehci_Qh[ USB_EHCI_NUM_QH ];
#pragma location="EHCI_QTD"
static USB_EHCI_QTD             ehci_Qtd[ USB_EHCI_NUM_QTD ];
#pragma location="EHCI_ITD"
static USB_EHCI_ITD             ehci_Itd[ USB_EHCI_NUM_ITD ];
#pragma location="EHCI_SITD"
static USB_EHCI_SITD            ehci_Sitd[ USB_EHCI_NUM_SITD ];
#endif  /* __ICCARM__ */
```

r_usb_hOhciMemory.c

```
#ifdef __ICCARM__
#pragma location="OHCI_HCCA"
static USB_OHCI_HCCA_BLOCK                  ohci_hcca;
#pragma location="OHCI_TD"
static USB_OHCI_HCD_TRANSFER_DESCRIPTOR    ohci_TdMemory[USB_OHCI_NUM_TD];
#pragma location="OHCI_ED"
static USB_OHCI_HCD_ENDPOINT_DESCRIPTOR    ohci_EdMemory[USB_OHCI_NUM_ED];
#endif  /* __ICCARM__ */
```

DS-5

DS-5 sets the section in the linker setting file (scatter file).

Changes are as follows:

・Start address of RAM region from 0x00040000and BSS region(0clear init memory region) follow RAM region.

```
    DATA                0x00040000    UNINIT
    {
        * (+RW)
    }
    BSS                 +0
    {
        * (+ZI)
    }
```

・To the EHCI and OHCI to fixed address, adds memory region definition.

```
        EHCI_PERIODIC_FRAMELIST 0x00020000    0x400
        {
                r_usb_hEhciMemory.o(EHCI_PFL)
        }
        EHCI_QTD         +0
        {
                r_usb_hEhciMemory.o(ehci_Qtd)
        }
        EHCI_ITD         +0
        {
                r_usb_hEhciMemory.o(ehci_Itd)
        }
        EHCI_QH          +0
        {
                r_usb_hEhciMemory.o(ehci_Qh)
        }
        EHCI_SITd        +0
        {
                r_usb_hEhciMemory.o(ehci_Sitd)
        }
        OHCI_HCCA        0x0003A000      0x100
        {
                r_usb_hOhciMemory.o(OHCI_HCCA)
        }
        OHCI_TDMEMORY    +0
        {
                r_usb_hOhciMemory.o(OHCI_TD)
        }
        OHCI_EDMEMORY    +0
        {
                r_usb_hOhciMemory.o(OHCI_ED)
        }
```

Variable definitions in the code are as follows.

r_usb_hEhciMemory.c

```
#ifdef __CC_ARM
#pragma arm section zidata = "EHCI_PFL"
static uint32_t         ehci_PeriodicFrameList[ USB_EHCI_PFL_SIZE ];
#pragma arm section zidata
#pragma arm section zidata = "EHCI_QH"
static USB_EHCI_QH      ehci_Qh[ USB_EHCI_NUM_QH ];
#pragma arm section zidata
#pragma arm section zidata = "EHCI_QTD"
static USB_EHCI_QTD     ehci_Qtd[ USB_EHCI_NUM_QTD ];
#pragma arm section zidata
#pragma arm section zidata = "EHCI_ITD"
static USB_EHCI_ITD     ehci_Itd[ USB_EHCI_NUM_ITD ];
#pragma arm section zidata
#pragma arm section zidata = "EHCI_SITD"
static USB_EHCI_SITD    ehci_Sitd[ USB_EHCI_NUM_SITD ];
#pragma arm section zidata
#endif
```

r_usb_hOhciMemory.c

```
#ifdef __CC_ARM
#pragma arm section zidata = "OHCI_HCCA"
static USB_OHCI_HCCA_BLOCK                  ohci_hcca;
#pragma arm section zidata
#pragma arm section zidata = "OHCI_TD"
static USB_OHCI_HCD_TRANSFER_DESCRIPTOR    ohci_TdMemory[USB_OHCI_NUM_TD];
#pragma arm section zidata
#pragma arm section zidata = "OHCI_ED"
static USB_OHCI_HCD_ENDPOINT_DESCRIPTOR    ohci_EdMemory[USB_OHCI_NUM_ED];
#pragma arm section zidata
#endif
```

Call the USB-BASIC-FW function

Adds the usbh_main() of USB-BASIC-F/W in the main() of "\src\sample\int_main.c".

```
extern void usbh_main(void);

int main (void)
{
    /* Initialize the port function */
    port_init();

    /* Initialize the ECM function */
    ecm_init();

    /* Initialize the ICU settings */
    icu_init();

    /* USBh main */
    usbh_main();

    while (1)
    {
        /* Toggle the PF7 output level(LED0) */
        PORTF.PODR.BIT.B7 ^= 1;

        soft_wait();  // Soft wait for blinking LED0

    }

}
```

Website and Support

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

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

Revision History

		Description	
Rev.	Date	Page	Summary
1.00	Aug 21, 2015	—	First edition issued
1.10	Dec 25, 2015	29	Added Appendix A
1.20	Feb 29, 2016	31,35,36	Added DS-5 setting
1.30	Dec 07, 2017	—	Corresponds to RZ / T1 initial setting Ver 1.30

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.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

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

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

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

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

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

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

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

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL II Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

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

Colophon 6.0