EC-1 Series

Peripheral Driver Manual

R01AN3581EJ0120
Rev.1.20
Sep 17, 2018

Outline

This application note explains the drivers and sample software used in the EC-1 series.

Target Devices

EC-1

Contents

1. Overview

To facilitate and speed up software development, the EC-1 microcontroller provides sample software to show the usage examples of each function.

This document describes the specifications for the driver for each EC-1 function and behavior of sample software.

1.1 Configuration

The figure below shows the layered structure of sample software.

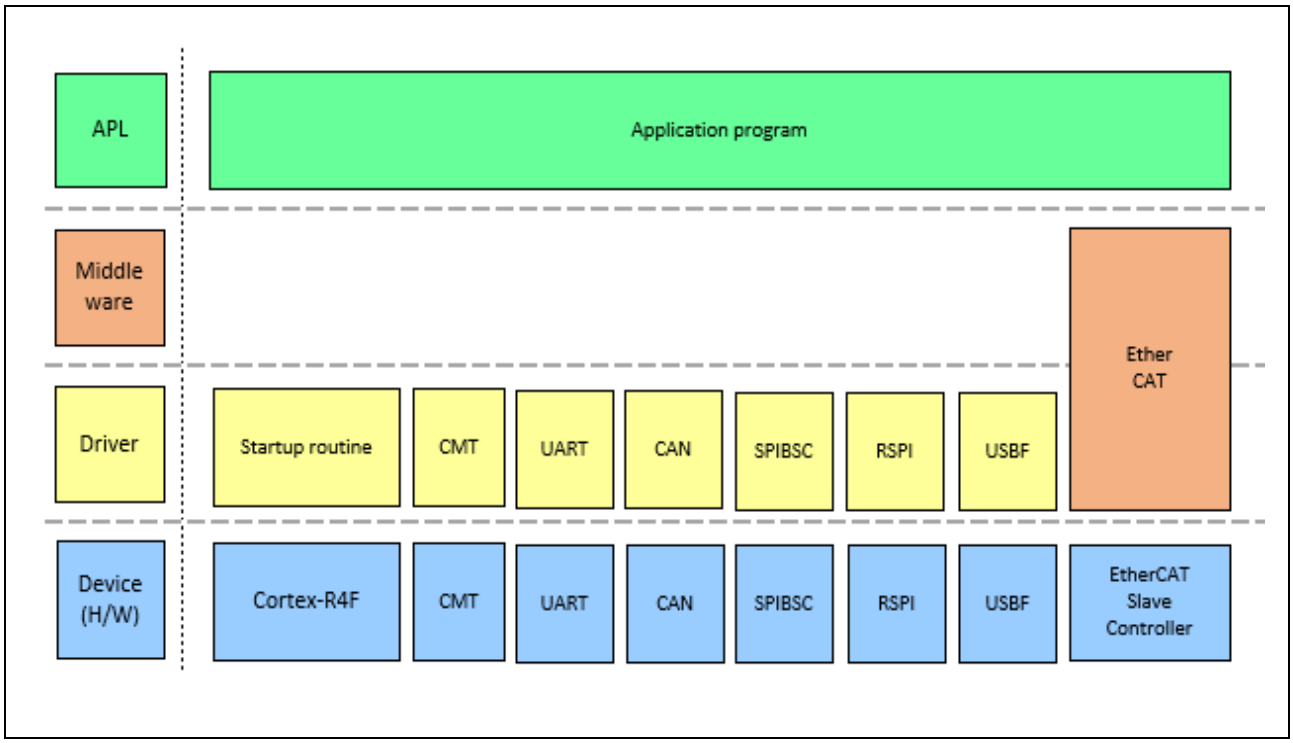


Figure 1-1 Layered Structure of Sample Software

1.2 Development Environment

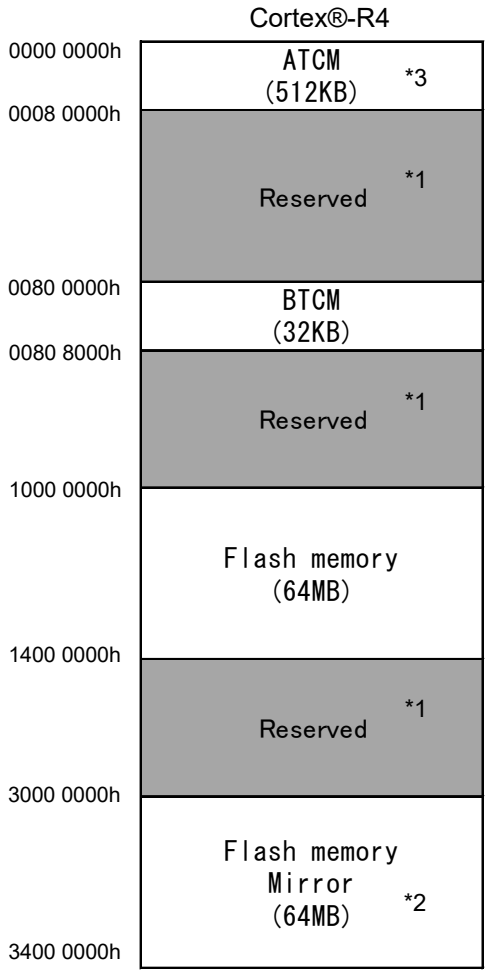
This section describes the software development tool.

This sample software is based on CMSIS V2.10. For details, see documents about CMSIS.

Table 1-1 Software Development Tools (Tool Chain)

Tool Chain	IDE	Compiler	Debugger	ICE
IAR	Embedded Workbench for ARM V7.70.1 (IAR Systems)	Embedded Workbench for ARM V7.70.1 (IAR Systems)	Embedded Workbench for ARM V7.70.1 (IAR Systems)	I-jet JTAGjet-Trace-CM (IAR Systems)
GCC	Renesas e2studio	GNUARM-NONE v16.01-EABI	Renesas e2studio	Segger J-Link ARM

1.3 Memory Allocation



Note 1. Do not access the reserved area.

Note 2. To be cached (for the other areas, do not enable cache by the MPU).

Note 3. Cortex-R4 tightly-coupled memory space

Figure 1-2 Memory Map

1.4 Program Allocation Example

The figure below shows a program allocation example at the time when serial flash ROM is booted.

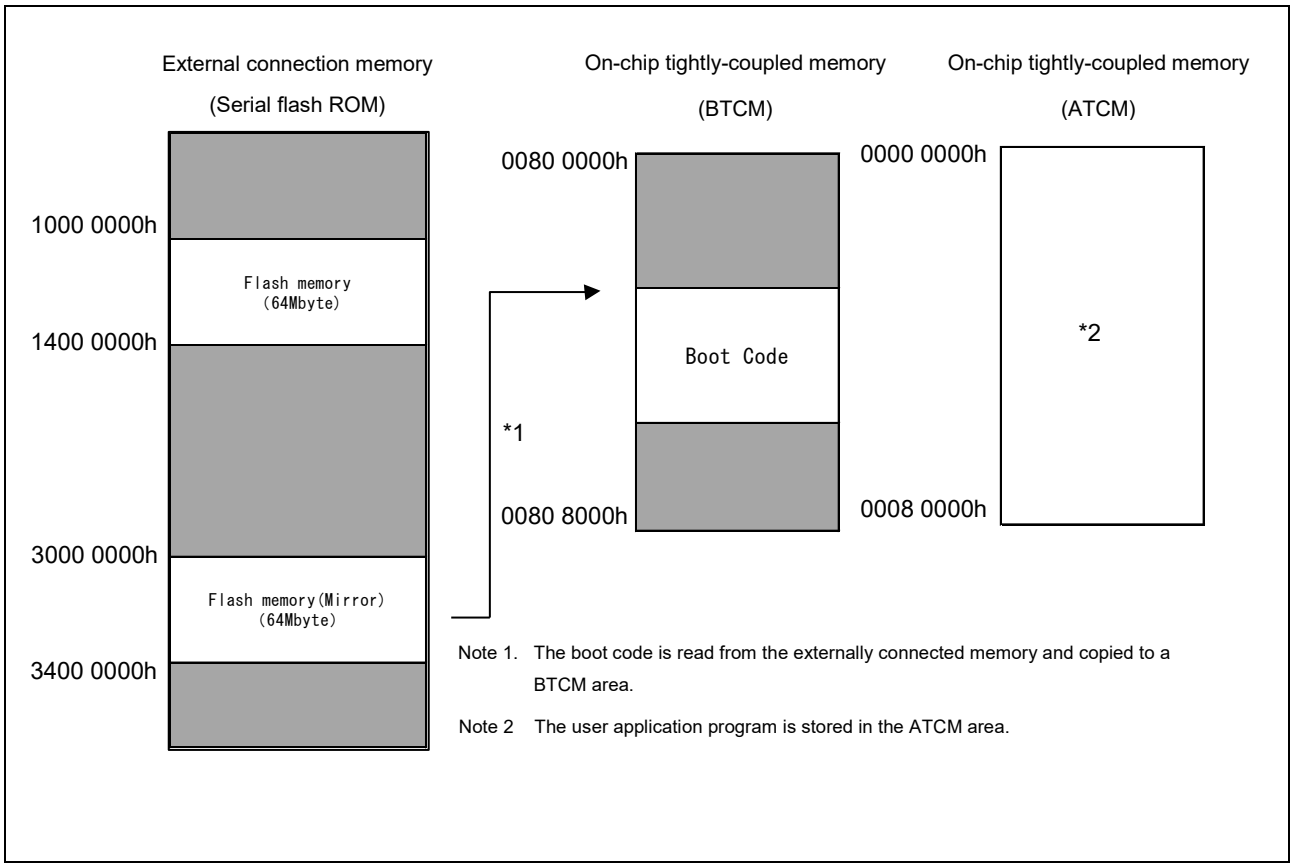


Figure 1-3 Program Allocation Example

For details, see section 3.4, Operating Mode, in the EC-1 User's Manual: Hardware.

2. File Structure

This section describes the directory structure and file structure of sample software.

2.1 Directory Structure

Table 2-1 Directory Structure of Sample Software

Directory	Description
./	Directory for storing sample software
./Include	Directory for storing the include files
./Library	Directory for storing libraries
./Source	Directory for storing sources

2.2 ./Include: Include Files

The table below shows the file structure of include files.

Table 2-2 File Structure of the Include File Directory

Directory	File	Description
board/	board.h	EC-1 CPU board configuration
can/	can_reg.h	CAN settings
	r_can_api.h	API prototype declaration for the CAN sample program
	r_can_config.h	CAN configuration
cmt/	r_cmt.h	Prototype declaration of the CMT driver
eth/	r_ether.h	Prototype declaration of the ETHER driver
rspi/	r_rspi.h	Prototype declaration of the RSPI driver
scifa/	r_scifa_api.h	API prototype declaration for the CAN sample program
	r_scifa_uart.h	Prototype declaration of the SCIFA_UART driver
	sio_char.h	Serial I/O configuration for char-type control
sflash/	r_sflash.h	Prototype declaration of the serial flash ROM driver
usbf/	r_usb_basic_config.h	Basic user definition
	r_usb_basic_if.h	Basic interface definition
	r_usb_cdefusbip.h	IP configuration
	r_usb_pcdc_config.h	pcdc user definition
	r_usb_pcdc_if.h	pcdc interface definition
wdta/	r_wdt_config.h	WDTA user definition
	r_wdt_if.h	WDTA interface definition
iwdta/	r_iwdt_config.h	IWDTA user definition
	r_iwdt_if.h	IWDTA interface definition
riic/	r_riic_ec1_config.h	RIIC user definition
	r_riic_ec1_if.h	RIIC interface definition
usbh/	r_usb_basic_config.h	USBh BASIC user definition
	r_usb_basic_if.h	USBh BASIC interface definition
	r_usb_hatapi_define.h	USB Common external header
	r_usb_hmsc_config.h	HMSC user definition
	r_usb_hmsc_if.h	HMSC interface definition

Directory	File	Description
./	iodefine.h	Register definition
	platform.h	MCU board configuration
	r_atcm_init.h	API header for setting the ATCM access wait
	r_bsc.h	API header for setting BSC
	r_cpg.h	API header for setting CPG
	r_ecm.h	API header for setting ECM
	r_icu_init.h	Initializing the interrupt controller unit
	r_mpc.h	API header for setting MPC
	r_port.h	API header for setting ports
	r_reset.h	API for resetting the EC-1 and low-power API header
	r_system.h	System configuration

2.3 ./Library: Libraries

This directory includes no files.

2.4 ./Source: Sources

The table below shows the structure of the source directory.

Table 2-3 Structure of Source Directory

Directory	Description
Driver	Driver-related source files
Project	Sample application
Templates	Startup files, etc.

2.4.1 ./Source/Driver: Driver-Related Source Files

The table below shows the structure of source files related to drivers.

Table 2-4 File Structure of the Driver-Related Directory

Directory	File	Description
can/	r_can_api.c	CAN driver
cmt/	cmt_userdef.c	CMT user definition
	r_cmt.c	CMT driver
ether/	r_ether.c	ETHER driver
rspi/	r_rspi.c	RSPI driver
	r_rspi_user.c	RSPI user definition
scifa_uart/	r_scifa_api.c	API for the CAN sample program
	scifa_uart.c	SCIFA_UART driver
	scifa_uart_userdef.c	SCIFA_UART user definition
	siochar.c	Input/output API and SCIFA initialization
	siorw.c	Input/output control API
sflash/	r_sflash.c	Serial flash ROM driver
usbf/pcdc/	r_usb_pcdc_api.c	USB PCDC API
	r_usb_pcdc_driver.c	USB PCDC driver
	r_usb_pcdc_local.h	USB PCDC header
usbf/pcdc/utilities/	CDC_Demo.inf	CDC device driver configuration

Directory	File	Description
usbf/basic/	r_usb_cdataio.c	Low-level I/O code
	r_usb_cextern.h	Common extern code
	r_usb_cintfifo.c	Interrupt code
	r_usb_cinthandler_usbip0.c	Interrupt handler code
	r_usb_clibusbip.c	Low-level library
	r_usb_creg_abs.c	USB register access
	r_usb_creg_access.c	USB IP register access
	r_usb_cusb_bitdefine.h	USB bit definition
	r_usb_dma.c	DMA configuration
	r_usb_dmac.h	DMA configuration header
	r_usb_pcontrolrw.c	Control transfer API
	r_usb_pdriver.c	USB driver
	r_usb_pdriverapi.c	USB driver API
	r_usb_pintfifo.c	FIFO access
	r_usb_preg_abs.c	Register access
	r_usb_preg_access.c	Signal code
	r_usb_psignal.c	Signal control code
	r_usb_pstdfunction.c	Standard function code
	r_usb_pstdrequest.c	Standard request code
	r_usb_reg_access.h	Register access header
	ec1_mcu.c	EC-1 configuration processing
wdta/	r_wdt.c	WDTA driver
iwdta/	r_iwdt.c	IWDTA driver
riic/	r_riic_ec1.c	RIIC driver
	r_riic_ec1_private.h	RIIC driver header
usbh/basic/	r_usb_basic_local.h	USB BASIC header
	r_usb_cscheduler.c	USB Host Scheduler
	r_usb_hDriver.c	USB host control driver
	r_usb_hdriverapi.c	USB host control driver API
	r_usb_hEhciDefUsr.h	EHCI User Defined Header
	r_usb_hEhciExtern.h	EHCI Extern header
	r_usb_hEhciMain.c	EHCI main processing
	r_usb_hEhciMemory.c	EHCI memory control code
	r_usb_hEhciTransfer.c	EHCI transfer code
	r_usb_hEhciTypedef.h	EHCI type definition header
	r_usb_hHci.c	HCI common code
	r_usb_hHci.h	HCI header
	r_usb_hHciLocal.h	HCI local common header
	r_usb_hhubsys.c	USB host hub system code
	r_usb_hManager.c	USB host control manager
	r_usb_hOhciDefUsr.h	OHCI User Defined Header
	r_usb_hOhciExtern.h	OHCI Extern header
	r_usb_hOhciMain.c	OHCI main process
	r_usb_hOhciMemory.c	OHCI memory control code
	r_usb_hOhciTransfer.c	OHCI transfer code
	r_usb_hOhciTypedef.h	OHCI type definition header

2.4.2 ./Source/Project: Sample Applications

The table below shows the structure of sample applications.

Table 2-5 File Structure of the Sample Application Directory

Directory	File	Description
can_sample/	main.c	CAN sample main processing source file
can_sample/IAR/	EC_1_can_serial_boot.eww	IAR project file
	EC_1_can_serial_boot.ewd	IAR project-related file
	EC_1_can_serial_boot.ewp	IAR project-related file
	EC_1_can_ram_debug.eww	IAR project file
	EC_1_can_ ram_debug.ewd	IAR project-related file
	EC_1_can_ ram_debug.ewp	IAR project-related file
can_sample/GCC/	EC-1_e2sws_serial_boot.bat	Project boot batch file
	serial_boot_sample.zip	Project archive file
rspi_sample/	main.c	RSPI sample main processing source file
	board_RenesaEva.c	Board configuration file
rspi_sample/IAR/	EC-1_rspi_serial_boot.eww	IAR project file
	EC-1_rspi_serial_boot.ewd	IAR project-related file
	EC-1_rspi_serial_boot.ewp	IAR project-related file
	EC-1_rspi_ram_debug.eww	IAR project file
	EC-1_rspi_ram_debug.ewd	IAR project-related file
	EC-1_rspi_ram_debug.ewp	IAR project-related file
rspi_sample/GCC/	EC-1_e2sws_serial_boot.bat	Project boot batch file
	serial_boot_sample.zip	Project archive file
sflash_writer_sample/	main.c	sflash_writer sample main processing source file
	flash_writer.c	flash_writer source file
	flash_writer.h	flash_writer header file
sflash_writer_sample/IAR/	EC_1_sflash_writer_serial_boot.eww	IAR project file
	EC_1_sflash_writer_serial_boot.ewd	IAR project-related file
	EC_1_sflash_writer_serial_boot.ewp	IAR project-related file
	EC_1_sflash_writer_ram_debug.eww	IAR project file
	EC_1_sflash_writer_ram_debug.ewd	IAR project-related file
	EC_1_sflash_writer_ram_debug.ewp	IAR project-related file
sflash_writer_sample/GCC/	EC-1_e2sws_serial_boot.bat	Project boot batch file
	serial_boot_sample.zip	Project archive file
usbf_sample/	main.c	USBF sample main processing source file
	r_usb_pcdc_apl.c	PCDC API
	r_usb_pcdc_descriptor.c	PCDC configuration data
usbf_sample/IAR/	EC_1_usbf_serial_boot.eww	IAR project file
	EC_1_usbf_serial_boot.ewd	IAR project-related file
	EC_1_usbf_serial_boot.ewp	IAR project-related file
	EC_1_usbf_ram_debug.eww	IAR project file
	EC_1_usbf_ram_debug.ewd	IAR project-related file
	EC_1_usbf_ram_debug.ewp	IAR project-related file
usbf_sample/GCC/	EC-1_e2sws_serial_boot.bat	Project boot batch file
	serial_boot_sample.zip	Project archive file

Directory	File	Description
wdta_sample/	main.c	WDTA sample main processing source file
wdta_sample/IAR/	EC_1_wdta_serial_boot.eww	IAR project file
	EC_1_wdta_serial_boot.ewd	IAR project-related file
	EC_1_wdta_serial_boot.ewp	IAR project-related file
	loader_init_sflash.c	EC-1 peripheral setting initialization file
wdta_sample/GCC/	EC-1_e2sws_serial_boot.bat	Project boot batch file
	serial_boot_sample.zip	Project archive file
	loader_init_sflash.c	EC-1 peripheral setting initialization file
iwdta_sample/	main.c	IWDTA sample main processing source file
iwdta_sample/IAR/	EC_1_iwdta_serial_boot.eww	IAR project file
	EC_1_iwdta_serial_boot.ewd	IAR project-related file
	EC_1_iwdta_serial_boot.ewp	IAR project-related file
	loader_init_sflash.c	EC-1 peripheral setting initialization file
iwdta_sample/GCC/	EC-1_e2sws_serial_boot.bat	Project boot batch file
	serial_boot_sample.zip	Project archive file
	loader_init_sflash.c	EC-1 peripheral setting initialization file
riic_sample/	main.c	RIIC sample main processing source file
riic_sample/IAR/	EC_1_riic_serial_boot.eww	IAR project file
	EC_1_riic_serial_boot.ewd	IAR project-related file
	EC_1_riic_serial_boot.ewp	IAR project-related file
	EC_1_riic_ram_debug.eww	IAR project file
	EC_1_riic_ram_debug.ewd	IAR project-related file
	EC_1_riic_ram_debug.ewp	IAR project-related file
riic_sample/GCC/	EC-1_e2sws_serial_boot.bat	Project boot batch file
	serial_boot_sample.zip	Project archive file
usbh_sample/	main.c	USBh sample main processing source file
	r_usb_hmsc_api.c	USBh MSC application code
	r_usb_hmsc_api.h	USB MSC application code header
	r_usb_main.c	USBh sample main processing source file
usbh_sample/GCC/	EC-1_e2sws_serial_boot.bat	Project boot batch file
	serial_boot_sample.zip	Project archive file
	EC-1_init_serial_boot.gsi	Linker script file for USBh
usbh_samole/IAR/	EC_1_usbh_serial_boot.eww	IAR project file
	EC_1_usbh_serial_boot.ewd	IAR project-related file
	EC_1_usbh_serial_boot.ewp	IAR project-related file
	EC_1_usbh_ram_debug.eww	IAR project file
	EC_1_usbh_ram_debug.ewd	IAR project-related file
	EC_1_usbh_ram_debug.ewp	IAR project-related file
	EC-1_init_ram_debug.icf	Mapping file
	EC-1_init_serial_boot.icf	Mapping file

2.4.3 ./Source/Templates: Startup Files, Etc.

The table below shows the structure of source files including startup files, etc.

Table 2-6 File Structure of the Startup-Related Directory

Directory	File	Description
Templates/	exit.c	Exit sequence
	r_atcm_init.c	API for setting the ATCM access wait
	r_cpg.c	API for setting CPG
	r_ecm.c	API for setting ECM
	r_icu_init.c	Initializing EC-1 device settings
	r_mpc.c	API for setting MPC
	r_reset.c	API for resetting the EC-1 and low-power API
Templates/IAR/	loader_init.asm	Interrupt service routine for the EC-1
	vector.asm	Vector table configuration
Templates/IAR/serial_boot/	bus_init_serial_boot.c	Initializing bus settings
	EC-1_init_serial_boot.icf	Mapping file
	EC1_init_boot.mac	Initialization macro file
	loader_init_sflash.c	Initializing the EC-1 peripheral configuration
	loader_param_serial_boot.c	Setting parameters for SPI boot mode
Templates/IAR/ram_debug/	EC-1_init_ram_debug.icf	Mapping file
	EC1_init_ram_debug.mac	Initialization macro file
	loader_init_ram.c	Initializing the EC-1 peripheral configuration
Templates/GCC/	loader_init.asm	Interrupt service routine for the EC-1
	vector.asm	Vector table configuration
Templates/GCC/ram_debug/	loader_init_ram.c	Initializing the EC-1 peripheral configuration
Templates/GCC/serial_boot/	bus_init_serial_boot.c	Initializing bus settings
	loader_init_sflash.c	Initializing the EC-1 peripheral configuration
	loader_param_serial_boot.c	Setting parameters for SPI boot mode
	EC-1_init_serial_boot.gsi	Linker script file

3. Drivers

This section describes driver structures and functions.

3.1 Structures, Unions and Enumerated Types

This section lists the structures, unions and enumerated types defined for drivers.

3.1.1 CAN Control

Table 3-1 can_vector_t Structure

Member Name		Description
uint8_t BYTE		Frame information
	uint8_t CANGE :1	CAN global error interrupt
	uint8_t CANIE0:1	CAN0 error interrupt
	uint8_t CANIE1:1	CAN1 error interrupt
	uint8_t CANRFI:1	CAN reception FIFO Interrupt
	uint8_t CANFIR0:1	CAN0 transmission/reception FIFO interrupt
	uint8_t CANTI0:1	CAN0 transmission interrupt
	uint8_t CANFIR1:1	CAN1 transmission/reception FIFO interrupt
	uint8_t CANTI1:1	CAN1 transmission interrupt

Table 3-2 can_callback_t Structure

Member Name	Description
void (*pintr_ge)(void);	Pointer to user callback function
void (*pintr_ie0)(void);	Pointer to user callback function
void (*pintr_ie1)(void);	Pointer to user callback function
void (*pintr_rfi)(void);	Pointer to user callback function
void (*pintr_fir0)(void);	Pointer to user callback function
void (*pintr_ti0)(void);	Pointer to user callback function
void (*pintr_fir1)(void);	Pointer to user callback function
void (*pintr_ti1)(void);	Pointer to user callback function

Table 3-3 can_handle_t Structure

Member Name	Description
bool	ch_opened
can_callback_t	can_callback

Table 3-4 can_rx_rule_t Structure_Reception Rule Table Structure

Member Name	Description
uint32_t buf_type;	Type of the buffer to be used Transmission: CAN_TX_BUFFER, CAN_TX_FIFO Reception: CAN_RX_BUFFER, CAN_RX_RX_FIFO, CAN_RX_FIFO
uint32_t rule_page;	Reception rule page number
uint32_t rule_table;	Reception rule table number
uint32_t rule_id;	Message ID
uint32_t rule_type;	Message type (data frame/remote frame)
uint32_t rule_format;	Message format (Standard ID/Extended ID)
uint32_t rule_label;	Message label
uint32_t rule_dlc_check;	DLC checking
uint32_t rule_mask;	Mask

Table 3-5 udata_t Structure_4-Byte Data Union

Member Name	Description
uint32_t LONG;	-
uint8_t BYTE(4);	-

Table 3-6 CAN_tx_message_t Structure_Message for Transmission Data Structure

Member Name	Description
uint32_t id;	Message ID
uint32_t type;	0: Data frame / 1: Remote frame
uint32_t format;	0: Standard ID / 1: Extended ID
uint32_t length;	Message data length
udata_t data_h;	Message data
udata_t data_l;	Message data
uint32_t history_label;	Label
uint32_t buf_type;	type of the buffer to be used (buffer or FIFO)

Table 3-7 CAN_rx_message_t Structure_Received Message Data Structure

Member Name	Description
uint32_t id;	Message ID
uint32_t format;	0: Standard ID / 1: Extended ID
uint32_t type;	0: Data frame / 1: Remote frame
uint16_t timestamp;	Timestamp data
uint16_t label;	Label information
uint32_t length;	Message length
udata_t data_h;	Message data
udata_t data_l;	Message data

Table 3-8 CAN_rx_message_t Structure_Global Error Source Count Structure

Member Name	Description
uint32_t total_num;	Total count of the global errors
uint32_t dlc_error_num;	DLC error count
uint32_t fifo_message_lost_num;	FIFO message loss count
uint32_t history_overflow_num;	Transmission history overflow count

Table 3-9 ch_err_source_t Structure_Channel Error Source Count Structure

Member Name	Description
uint32_t total_num;	Total count of the channel errors
uint32_t bus_error_num;	Bus error count
uint32_t error_warning_num;	Error warning count
uint32_t error_passive_num;	Error passive count
uint32_t bus_off_start_num;	Bus-off start count
uint32_t bus_off_return_num;	Bus-off return count
uint32_t overload_num;	Overload count
uint32_t bus_lock_num;	Bus lock-up count
uint32_t arbitration_lost_num;	Arbitration loss count
uint32_t staff_error_num;	Staff error count
uint32_t form_error_num;	Form error count
uint32_t ack_error_num;	ACK error count
uint32_t crc_error_num;	CRC error count
uint32_t recessive_bit_error_num;	Recessive bit error count
uint32_t dominant_bit_error_num;	Dominant bit error count
uint32_t ack_delimiter_error_num;	ACK delimiter error count

Table 3-10 ch_tx_source_t Structure_Transmission Interrupt Source Count Structure

Member Name	Description
uint32_t total_num;	Total count of the transmission interrupts
uint32_t intr_sorce;	Interrupt source
uint32_t tx_buf_end_num;	Count of successful transmissions from the transmission buffer
uint32_t tx_fifo_end_num;	Count of successful transmissions from the FIFO buffer
uint32_t tx_abort_num;	Transmission abortion count
uint32_t tx_queue_num;	Transmission queue count
uint32_t tx_history_num;	Transmission history data count

Table 3-11 CAN_intr_source_t Structure_Interrupt Source Information Structure

Member Name	Description
uint32_t rx_fifo_num;	Reception FIFO interrupt count
uint32_t ch_fifo_receive_num;	Transmission/reception FIFO receive interrupt count
ch_tx_source_t tx_source;	Transmission interrupt source
gl_err_source_t gl_err;	Global error interrupt source
ch_err_source_t ch_err;	Channel error interrupt source

Table 3-12 can_used_buffer_t Structure_Used Buffer Information Structure

Member Name	Description
uint32_t use_tx_buf_no;	Transmission buffer number
uint32_t use_rx_buf_no;	Reception buffer number
uint32_t use_rx_fifo_no;	Reception FIFO buffer number
uint32 use_fifo_txmode_no;	Number of the transmission/reception FIFO buffer in transmission mode
uint32_t use_fifo_rxmode_no;	Number of the transmission/reception FIFO buffer in reception mode
uint32_t use_fifo_link_buf_no;	Number of the buffer to which the FIFO buffer is linked to

Table 3-13 can_tx_intr_sts_t Structure_Transmission Interrupt Request Status Information Structure

Member Name		Description
uint8_t BYTE		Frame information
	uint8_t TMTRF:2;	Result of transmission from the transmission buffer
	uint8_t CFTXIF:1;	Request for transmission/reception FIFO transmission interrupt
	uint8_t TXQIF:1;	Request for transmission queue interrupt
	uint8_t THLIF:1;	Request for transmission history interrupt
	uint8_t :3;	-

Table 3-14 can_tx_history_t Structure_Transmission History Information Structure

Member Name	Description
uint32_t buf_type;	Buffer type
uint32_t buf_no;	Buffer number
uint32_t label;	Label data

3.1.2 RSPI Control

Table 3-15 rspi_state_t Structure

Member Name	Description		
uint8_t b_ovrf;	Overrun error flag	0: No error	1: Overrun error
uint8_t b_idlnf;	RSPI idle flag	0: Idle state	1: Not Idle state
uint8_t b_modf;	Mode fault error flag	0: No error	1: Mode fault error
uint8_t b_perf;	Parity error flag	0: No error	1: Parity error
uint8_t b_invalid;	invalid packet flag	0: No error	1: Invalid packet data
uint8_t b_rx_end;	Receive end flag	0: Not end	1: Recieve end
uint8_t b_tx_end;	Transmit end flag	0: Not end	1: Recieve end
uint8_t	not use		

3.1.3 SCIFA_UART Control

Table 3-16 scifa_callback_t Structure

Member Name	Description
void (*pintr_ski)(void);	Pointer to user callback function

Table 3-17 scifa_handle_t Structure

Member Name	Description
scifa_callback_t scifa_callback;	scifa_callback

Table 3-18 scifa_vector_para_t Structure

Member Name	Description
uint16_t vector_no;	Vector number
uint32_t priority;	Priority
void (_irq _arm * pcallback_isr)(void);	Pointer to user callback interrupt function

3.1.4 Serial Flash ROM Control

Table 3-19 crr_sflash_infot Structure

Member Name	Description
uint32_t id;	Serial flash ROM ID
uint32_t device_size;	Serial flash ROM device size
uint32_t erace_size;	Serial flash ROM erase size
uint32_t page_size;	Serial flash ROM program size

3.1.5 USB Function Control

Table 3-20 USB_REQUEST_t Structure

USB_REQUEST_t is a structure for storing USB requests except standard requests.

It is used as an argument of the callback function registered in ctrltrans of the USB_PCDREG_t structure.

Member Name	Description
uint16_t ReqType;	Contains the value of bmRequestType[D7-D0]. (Contains the value of the BMREQUESTTYPE bits in the USBREQ register.) During use, mask it with USB_BMREQUESTTYPE (0x00FFu).
uint16_t ReqTypeType;	Contains the value of Type[D6-D5] of bmRequestType. (Contains the value of bits 6 and 5 of the BMREQUESTTYPE bits in the USBREQ register.) During use, mask it with USB_BMREQUESTTYPETYPE (0x0060u).
uint16_t ReqTypeRecip;	Contains the value of Recipient[D4-D0] of bmRequestType. (Contains the value of bits 4 to 0 of the BMREQUESTTYPE bits in the USBREQ register.) During use, mask it with USB_BMREQUESTTYPERECIP (0x001Fu).
uint16_t ReqRequest;	Contains the value of bRequest. (Contains the value of the BREQUEST bits in the USBREQ register.) During use, mask it with USB_BREQUEST (0xFF00u).
uint16_t ReqValue;	Contains the value of wValue. (Contains the value of USBVAL register.)
uint16_t ReqIndex;	Contains the value of wIndex. (Contains the value of USBINDEX register.)
uint16_t ReqLength;	Contains the value of wLength. (Contains the value of USBLENG register.)

Table 3-21 USB_UTR_t Structure

USB_UTR_t is a structure used for data transfer except control transfer.

Member Name	Description
uint16_t keyword;	Register a pipe number used for data transfer.
USB_UTR_CB_t complete;	Register the function to be started when data transfer is completed.
void *tranadr;	Register the address of the buffer used for data transfer.
uint32_t tranlen;	Specify the data transfer size. The transfer size must be smaller than the size of the buffer for data transfer.
uint16_t status;	The results of data transfer are stored by the peripheral control driver (PCD). See section 7.13.2, Notification of Transfer Result.
uint16_t pipectr;	The value of the PIPECTR register is stored by the peripheral control driver (PCD).

Table 3-22 USB_PCDREG_t Structure

USB_PCDREG_t is a structure for registering information about the peripheral device class driver (PDCD).

The callback function registered in USB_PCDREG_t is executed upon a change of the device state or other sources.

Member Name	Description
uint16_t **pipetbl;	Address of the pipe information table
uint8_t *devicetbl;	Address of the Device Descriptor table
uint8_t *qualitbl;	Address of the Device Qualifier Descriptor table
uint8_t **configtbl;	Address of the Configuration Descriptor table
uint8_t **othertbl;	Address of the Other Speed Descriptor table
uint8_t **stringtbl;	Address of the String Descriptor table
USB_CB_t devdetach;	Register the function to be started when entering the default state. The registered function is called when detecting a USB reset.
USB_CB_t devconfig;	Register the function to be started when entering the configured state. The registered function is called in the SET_CONFIGURATION request status stage.
USB_CB_t devdetach;	Register the function to be started when entering the detached state. The registered function is called when detecting a detached state.
USB_CB_t devsuspend;	Register the function to be started when entering the suspended state. The registered function is called when detecting a suspended state
USB_CB_t devresume;	Register the function to be started when entering the resume state. The registered function is called when detecting a resume state.
USB_CB_t interface;	Register the function to be started when changing the interface. The registered function is called in the SET_INTERFACE request status stage.
USB_CB_TRN_t ctrltrans;	Register the function to be started when receiving a USB request except standard commands.

Table 3-23 USB_SCI_SerialState_t Structure

Member Name		Description
unsigned long WORD		-
	uint16_t bRxCarrier:1;	Data Carrier Detect: The carrier was detected on the line.
	uint16_t bTxCarrier:1;	Data Set Ready: The line is connected and ready for communication.
	uint16_t bBreak:1;	A break signal was detected.
	uint16_t bRingSignal:1;	A ring signal was sensed.
	uint16_t bFraming:1;	A framing error was detected.
	uint16_t bParity:1;	A parity error was detected.
	uint16_t bOverRun:1;	An overrun error was detected.
	uint16_t rsv:9;	Reserved

3.1.6 WDTA Control

Table 3-24 wdt_err_t Enumeration Type

Member Name	Description
WDT_SUCCESS = 0	-
WDT_ERR_OPEN_IGNORED	The module has already been Open()ed
WDT_ERR_INVALID_ARG	Argument is not valid for parameter
WDT_ERR_NULL_PTR	Received null pointer or missing required argument
WDT_ERR_NOT_OPENED	Open function has not yet been called

Table 3-25 wdt_ch_t Enumeration Type

Member Name	Description
WDT_CHANNEL_0 = 0	ch0
WDT_CHANNEL_1	ch1
WDT_CHANNEL_MAX	-

Table 3-26 wdt_timeout_t Enumeration Type

Member Name	Description
WDT_TIMEOUT_1024 = 0x0000u	1024 (cycles)
WDT_TIMEOUT_4096 = 0x0001u	4096 (cycles)
WDT_TIMEOUT_8192 = 0x0002u	8192 (cycles)
WDT_TIMEOUT_16384 = 0x0003u	16,384 (cycles)
WDT_NUM_TIMEOUTS	-

Table 3-27 wdt_clock_div_t Enumeration Type

Member Name	Description
WDT_CLOCK_DIV_4 = 0x0010u	WDTCLK/4
WDT_CLOCK_DIV_64 = 0x0040u	WDTCLK/64
WDT_CLOCK_DIV_128 = 0x00F0u	WDTCLK/128
WDT_CLOCK_DIV_512 = 0x0060u	WDTCLK/512
WDT_CLOCK_DIV_2048 = 0x0070u	WDTCLK/2048
WDT_CLOCK_DIV_8192 = 0x0080u	WDTCLK/8192

Table 3-28 wdt_window_end_t Enumeration Type

Member Name	Description
WDT_WINDOW_END_75 = 0x0000u	75%
WDT_WINDOW_END_50 = 0x0100u	50%
WDT_WINDOW_END_25 = 0x0200u	25%
WDT_WINDOW_END_0 = 0x0300u	0% (window end position is not specified)

Table 3-29 wdt_window_start_t Enumeration Type

Member Name	Description
WDT_WINDOW_START_25 = 0x0000u	25%
WDT_WINDOW_START_50 = 0x1000u	50%
WDT_WINDOW_START_75 = 0x2000u	75%
WDT_WINDOW_START_100 = 0x3000u	100% (window start position is not specified)

Table 3-30 wdt_timeout_control_t Enumeration Type

Member Name	Description
WDT_ERROR_ENABLE = 0x00u	Error output is enabled
WDT_ERROR_DISABLE = 0x80u	Error output is disabled

Table 3-31 wdt_config_t Structure

Member Name	Description
wdt_timeout_t timeout	Time-out period
wdt_clock_div_t wdtclk_div	WDT clock division ratio
wdt_window_start_t window_start	Window start position
wdt_window_end_t window_end	Window end position
wdt_timeout_control_t timeout_control	ERROR output when time-out

Table 3-32 wdt_cmd_t Enumeration Type

Member Name	Description
WDT_CMD_GET_STATUS	Get WDT status
WDT_CMD_REFRESH_COUNTING	Refresh the counter
WDT_CMD_NO_ACTION	-

3.1.7 IWDTA Control

Table 3-33 iwdt_err_t Enumeration Type

Member Name	Description
IWDT_SUCCESS = 0	-
IWDT_ERR_OPEN_IGNORED	The module has already been Open()ed
IWDT_ERR_INVALID_ARG	Argument is not valid for parameter
IWDT_ERR_NULL_PTR	Received null pointer or missing required argument
IWDT_ERR_NOT_OPENED	Open function has not yet been called

Table 3-34 iwdt_timeout_t Enumeration Type

Member Name	Description
IWDT_TIMEOUT_1024 = 0x0000u	1024 (cycles)
IWDT_TIMEOUT_4096 = 0x0001u	4096 (cycles)
IWDT_TIMEOUT_8192 = 0x0002u	8192 (cycles)
IWDT_TIMEOUT_16384 = 0x0003u	16,384 (cycles)
IWDT_NUM_TIMEOUTS	-

Table 3-35 iwdt_clock_div_t Enumeration Type

Member Name	Description
IWDT_CLOCK_DIV_1 = 0x0000u	WDTCLK/1
IWDT_CLOCK_DIV_16 = 0x0020u	WDTCLK/16
IWDT_CLOCK_DIV_32 = 0x0030u	WDTCLK/32
IWDT_CLOCK_DIV_64 = 0x0040u	WDTCLK/64
IWDT_CLOCK_DIV_12 8 = 0x00F0u	WDTCLK/128
IWDT_CLOCK_DIV_256 = 0x0050u	WDTCLK/256

Table 3-36 iwdt_window_end_t Enumeration Type

Member Name	Description
IWDT_WINDOW_END_75 = 0x0000u	75%
IWDT_WINDOW_END_50 = 0x0100u	50%
IWDT_WINDOW_END_25 = 0x0200u	25%
IWDT_WINDOW_END_0 = 0x0300u	0% (window end position is not specified)

Table 3-37 iwdt_window_start_t Enumeration Type

Member Name	Description
IWDT_WINDOW_START_25 = 0x0000u	25%
IWDT_WINDOW_START_50 = 0x1000u	50%
IWDT_WINDOW_START_75 = 0x2000u	75%
IWDT_WINDOW_START_100 = 0x3000u	100% (window start position is not specified)

Table 3-38 iwdt_timeout_control_t Enumeration Type

Member Name	Description
IWDT_ERROR_ENABLE = 0x00u	Error output is enabled
IWDT_ERROR_DISABLE = 0x80u	Error output is disabled

Table 3-39 iwdt_config_t Structure

Member Name	Description
iwdt_timeout_t timeout	Time-out period
iwdt_clock_div_t wdtclk_div	IWDT clock division ratio
iwdt_window_start_t window_start	Window start position
iwdt_window_end_t window_end	Window end position
iwdt_timeout_control_t timeout_control	ERROR output when time-out

Table 3-40 iwdt_cmd_t Enumeration Type

Member Name	Description
IWDT_CMD_GET_STATUS	Get IWDT status
IWDT_CMD_REFRESH_COUNTING	Refresh the counter
IWDT_CMD_NO_ACTION	-

3.1.8 RIIC Control

Table 3-41 riic_return_t Enumeration Type

Member Name	Description
RIIC_SUCCESS = 0U	Successful operation
RIIC_ERR_LOCK_FUNC	Lock has already been acquired by another task.
RIIC_ERR_INVALID_CHAN	None existent channel number
RIIC_ERR_INVALID_ARG	Parameter error
RIIC_ERR_NO_INIT	Uninitialized state
RIIC_ERR_BUS_BUSY	Channel is on communication.
RIIC_ERR_AL	Arbitration lost error
RIIC_ERR_OTHER	Other error

Table 3-42 riic_info_t Structure

Member Name	Description
uint8_t rsv2	Reserved
uint8_t rsv1	Reserved
riic_ch_dev_status_t dev_sts	Device status flag
uint8_t ch_no	Channel No.
riic_callback callbackfunc	Callback function
uint32_t cnt2nd	2nd data counter
uint32_t cnt1nd	1st data counter
uint8_t * p_data2nd	Pointer for 2nd data buffer
uint8_t * p_data1st	Pointer for 1st data buffer
uint8_t * p_slv_adr	Pointer for Slave address buffer

Table 3-43 riic_mcu_status_t Structure

Member Name		Description
uint32_t LONG		-
	uint32_t rsv:12	Reserve
	uint32_t AAS2:1	Slave2 address detection flag
	uint32_t AAS1:1	Slave1 address detection flag
	uint32_t AAS0:1	Slave0 address detection flag
	uint32_t GCA:1	Generalcall address detection flag
	uint32_t DID:1	DeviceID address detection flag
	uint32_t HOA:1	Host address detection flag
	uint32_t MST:1	Master mode / Slave mode flag
	uint32_t TMO:1	Time out flag
	uint32_t AL:1	Arbitration lost detection flag
	uint32_t SP:1	Stop condition detection flag
	uint32_t ST:1	Start condition detection flag
	uint32_t RBUF:1	Receive buffer status flag
	uint32_t SBUF:1	Send buffer status flag
	uint32_t SCLO:1	SCL pin output control status
	uint32_t SDAO:1	SDA pin output control status
	uint32_t SCLI:1	SCL pin level
	uint32_t SDAI:1	SDA pin level
	uint32_t NACK:1	NACK detection flag
	uint32_t TRS:1	Send mode / Receive mode flag
	uint32_t BSY:1	Bus status flag

3.1.9 USB Host Control

Table 3-44 USB_HCDREG_t Structure

The USB_HCDREG_t structure is used to register HDCD information.

Callback functions registered in the USB_HCDREG_t structure are executed when the device state changes.

Member Name	Description
uint16_t rootport	The connected port number is registered.
uint16_t devaddr	The device address is registered.
uint16_t devstate	The device connection status is registered.
uint16_t ifclass	Register the interface class code of HDCD.
uint16_t *tpl	Register the target peripheral list in which the HDCD operates. .
USB_CB_CHECK_t classcheck	Register the function to be executed when HDCD is checked. This function is called when TPL matches.
USB_CB_INFO_t devconfig	Register the function to be executed at the time of configured state transition. This function is called when the SET_CONFIGURATION request is completed.
USB_CB_INFO_t devdetach	Register the function to be executed at the time of detach state transition.
USB_CB_INFO_t devsuspend	Register the function to be executed at the time of suspended state transition.
USB_CB_INFO_t devresume	Register the function to be executed at the time of resume state transition.

Table 3-45 USB_SETUP_t Structure

The USB_SETUP_t structure is a data structure of USB request.

Member Name	Description
uint16_t type	Set bRequest[b15-b8] and bmRequestType[b7-b0].
uint16_t value	Set wValue.
uint16_t index	Set wIndex.
uint16_t length	Set wLength.
uint16_t devaddr	Device address

Table 3-46 USB_UTR_t Structure

The USB_UTR_t structure is used for USB communication and task messages.

USB communication with connected devices is made possible by setting this structure for the R_usb_hstd_TransferStart() argument.

Member Name	Description
uint16_t msginfo	Set the message information to be used by the USB-BASIC-F/W task. This setting is not required when USB communication is performed.
uint16_t keyword	Set the communication pipe number.
uint16_t result	The USB communication result is stored. See section 7.13.2 Notification of Transfer Result
USB_UTR_CB_t complete	Set the address of the function to be executed when USB communication is completed. See section 11.8.8 Callback Functions
void *tranadr	Set the USB communication buffer address. ・Reception or ControlRead transfer Set the address of the receive data buffer. ・Transmission or ControlWrite transfer Set the address of the transmit data buffer. ・NoDataControl transfer Even if the address is specified, it is ignored.
uint32_t tranlen	Set the USB communication data length. ・Reception or ControlRead transfer Specify the receive data length. This length is updated to a value (specified data length - actually received data length) at the end of USB communication. ・Transmission or ControlWrite transfer Specify the transmit data length. ・NoDataControl transfer Specify 0.
USB_SETUP_t *setup	Set the setup packet information for control transfer. See Table 3-45 USB_SETUP_t Structure and section 11.8.9(6) Control transfer

3.2 List of Constants and Error Codes

The table below lists constants and error codes provided for the driver.

3.2.1 CAN Control

Table 3-47 Constants Used by the CAN Driver

Constant Name	Setting Value	Description
CAN_NUM	2	Number of channels of the CAN module
CAN_CH_0	0	Channel 0 (CAN0)
CAN_CH_1	1	Channel 1 (CAN1)
CH_BUFFER_MAX	16	Number of transmission buffers available for each channel
CH_FIFO_BUFFER_MAX	3	Number of transmission/reception FIFO buffers available at each channel
DATA_MAX	8	Number of message data that can be transmitted at the same time
CAN_TX_BUFFER	0	State flag (the transmission buffers are in use)
CAN_TX_FIFO	1	State flag (the transmission/reception FIFO buffers are in use in transmission mode)
CAN_TX_HISTORY	2	State flag (for transmission history)
CAN_TX_QUEUE	3	State flag (for transmission queue)
CAN_RX_BUFFER	0	State flag (reception buffers are in use)
CAN_RX_RX_FIFO	1	State flag (reception FIFO buffers are in use)
CAN_RX_FIFO	2	State flag (transmission/reception FIFO buffers are in use in reception mode)
CAN_MODULE_ON	0	Exits the stop state
CAN_MODULE_OFF	1	Enters the stop state
CAN_STANDARD	0	Standard ID
CAN_EXTENDED	1	Extended ID
CAN_DATA_FRAME	0	Data frame
CAN_REMOTE_FRAME	1	Remote frame
CAN_RULE_PAGE_MAX	24	Maximum number of the reception rule pages
CAN_RULE_TABLE_MAX	16	Maximum number of the reception rule tables
CAN_RX_FIFO_BUFFER_MAX	8	Maximum number of the reception FIFO buffers
CAN_RX_BUFFER_MAX	32	Maximum number of the reception buffers
CAN_RULE_NUM_MAX	64	Maximum number of the reception rules
CAN_RX_MODE	0	Reception mode
CAN_TX_MODE	1	Transmission mode
CAN_GATEWAY_MODE	2	Gateway mode
CAN_FIFO_MSG_0	0	Number of the transmission/reception FIFO buffer stages (0 messages)
CAN_FIFO_MSG_4	1	Number of the transmission/reception FIFO buffer stages (4 messages)
CAN_FIFO_MSG_8	2	Number of the transmission/reception FIFO buffer stages (8 messages)
CAN_FIFO_MSG_16	3	Number of the transmission/reception FIFO buffer stages (16 messages)
CAN_FIFO_MSG_32	4	Number of the transmission/reception FIFO buffer stages (32 messages)
CAN_FIFO_MSG_48	5	Number of the transmission/reception FIFO buffer stages (48 messages)
CAN_FIFO_MSG_64	6	Number of the transmission/reception FIFO buffer stages (64 messages)
GL_MODE_STOP	0	Global stop mode
GL_MODE_RESET	1	Global reset mode
GL_MODE_TEST	2	Global test mode

Constant Name	Setting Value	Description
GL_MODE_OPE	3	Global operation mode
CAN_GL_OPE	0	Enters the global operating mode
CAN_GL_RESET	1	Enters the global reset mode
CAN_GL_TEST	2	Enters the global test mode
CH_MODE_STOP	0	Channel stop mode
CH_MODE_RESET	1	Chanel reset mode
CH_MODE_WAIT	2	Channel halt mode
CH_MODE_COMM	3	Channel transfer mode
CAN_CH_COMM	0	Enters the channel transfer mode
CAN_CH_RESET	1	Enters the channel reset mode
CAN_CH_WAIT	2	Enters the channel halt mode
GL_TEST_RAMTEST	0	RAM test
GL_TEST_COMMTEST	1	Inter-channel transfer test
CH_TEST_STANDARD	0	Standard test mode
CH_TEST_LISTENONLY	1	Listen-only mode
CH_TEST_SELF0	2	Self-test mode 0 (external loopback mode)
CH_TEST_SELF1	3	Self-test mode 1 (internal loopback mode)
CANCLKA_CLK	24000000u	CAN clock running at 24 MHz
CANCLKB_CLK	25000000u	CAN clock running at 25MHz
CAN_INTR_DISABLE	0	Interrupt is disabled
CAN_INTR_ENABLE	1	Interrupt is enabled
CAN_IR_PRIORITY_262_CANERR_GL	3	Priority order (CAN global error)
CAN_IR_PRIORITY_263_CANE RR_CH0	4	Priority order (CAN0 error)
CAN_IR_PRIORITY_264_CANE RR_CH1	4	Priority order (CAN1 error)
CAN_IR_PRIORITY_104_CANR FI	5	Priority order (CAN reception FIFO)
CAN_IR_PRIORITY_105_CANFI R0	5	Priority order (CAN0 transmission/reception FIFO reception completion)
CAN_IR_PRIORITY_106_CANTI 0	5	Priority order (CAN0 transmission)
CAN_IR_PRIORITY_107_CANFI R1	5	Priority order (CAN1 transmission/reception FIFO reception completion)
CAN_IR_PRIORITY_108_CANTI 1	5	Priority order (CAN1 transmission)
CAN_HVA_WRITE_DATA	0u	HVA write data
CAN_OK	0u	Returned value for successful operation
CAN_EMPTY	1u	Returned value in the case of buffer empty
CAN_NG	0xFFFFFFFFu	Returned value when an error occurred
CAN_INTR_TX_END	1	Source for the channel transmission interrupt: transmission completion
CAN_INTR_ABORT_END	2	Source for the channel transmission interrupt: abort transmission completion
CAN_INTR_FIFO_REQ	3	Source for the channel transmission interrupt: completion of transmission from the transmission/reception FIFO in transmission mode
CAN_INTR_QUEUE_REQ	4	Source for the channel transmission interrupt: transmission queue request is issued
CAN_INTR_HISTORY_REQ	5	Source for the channel transmission interrupt: transmission history request is issued
CAN_INTR_FIFO_EMPTY	1	Reception FIFO buffer empty
CAN_INTR_FIFO_FULL	2	Reception FIFO buffer full
CAN_INTR_FIFO_LOST	3	Reception FIFO buffer message lost
CAN_INTR_FIFO_TX_MESSAG E	4	Request for transmission/reception FIFO transmission interrupt
CAN_INTR_FIFO_RX_MESSAG E	5	Request for transmission/reception FIFO reception interrupt
CAN_BUS_ERR	1	Error flag (bus error)
CAN_ERR_WARNING	2	Error flag (error warning)

Constant Name	Setting Value	Description
CAN_ERR_PASSIVE	3	Error flag (error passive)
CAN_BUS_OFF_START	4	Error flag (entering the bus-off state)
CAN_BUS_OFF_RETURN	5	Error flag (recovery from the bus-off state)
CAN_OVER_LOAD	6	Error flag (overload)
CAN_BUS_LOCK	7	Error flag (channel bus lockup)
CAN_ARBITRATION_LOST	8	Error flag (arbitration lost)
CAN_STAFF_ERR	9	Error flag (staff error)
CAN_FORM_ERR	10	Error flag (form error)
CAN_ACK_ERR	11	Error flag (ACK error)
CAN_CRC_ERR	12	Error flag (CRC error)
CAN_RECESSIVE_BIT_ERR	13	Error flag (recessive bit error)
CAN_DOMINANT_BIT_ERR	14	Error flag (dominant bit error)
CAN_ACK_DELIMITER_ERR	15	Error flag (ACK delimiter error)
CAN_DLC_ERR	1	Error flag (DLC error)
CAN_FIFO_MSG_LOST_ERR	2	Error flag (FIFO message lost)
CAN_HISTORY_OVERFLOW_E RR	3	Error flag (transmission history buffer overflow)
CAN0_CRXD0_P30_VAL	0x10	MPC: setting value for the CAN0 CRXD0 (not used in this sample program)
CAN0_CRXD0_PC6_VAL	0x10	MPC: setting value for the CAN0 CRXD0
CAN0_CTXD0_P60_VAL	0x10	MPC: setting value for the CAN0 CTXD0 (not used in this sample program)
CAN0_CTXD0_P67_VAL	0x10	MPC: setting value for the CAN0 CTXD0
CAN1_CRXD1_PC3_VAL	0x10	MPC: setting value for the CAN1 CRXD1 (not used in this sample program)
CAN1_CRXD1_PC7_VAL	0x10	MPC: setting value for the CAN1 CRXD1
CAN1_CTXD1_P61_VAL	0x10	MPC: setting value for the CAN1 CTXD1 (not used in this sample program)
CAN1_CTXD1_P66_VAL	0x10	MPC: setting value for the CAN1 CTXD1
CAN1_CTXD1_PB3_VAL	0x10	MPC: setting value for the CAN1 CTXD1 (not used in this sample program)
CAN_GL_STATUS_BIT	0x00000007u	RSCAN0GSTS register mask bit
CAN_CH_STATUS_BIT	0x00000007u	RSCAN0CmSTS register mask bit
GCFG_REG_INIT	0x00000013u	Initial value for the RSCAN0GCFG register
TMIEC0_REG_DISABLE_LOW	0x0000FFFFu	TMIEp (p = 15 to 0) mask bit of the RSCAN0TMIEC0 register
TMIEC0_REG_DISABLE_HIGH	0xFFFF0000u	TMIEp (p = 31 to 16) mask bit of the RSCAN0TMIEC0 register
CAN_CH_STOP_MODE	0x00000004u	RSCAN0CmCTR register channel stop mode
CAN_REL_CH_STOP_MODE	0xFFFFFFFBu	Release the module from the RSCAN0CmCTR register channel stop mode
TIME_QUANTUM_MIN	8	Value range for the bit time*1 Set the value within the range obtained by SS + TSEG1 + TSEG2 = 8 to 25 Tq
TIME_QUANTUM_MAX	25	Value range for the bit time*1 Set the value within the range obtained by SS + TSEG1 + TSEG2 = 8 to 25 Tq
SAMPLE_POINT	0.666666667	Sample point (%)*1 The two thirds of one-bit communication frame is set as the sampling point in this sample program.
FIFO_UPDATE	0x000000FFu	Value for controlling the pointer to the FIFO buffers

Note 1. For details, refer to section 27.9.1.2, Bit Timing Setting, in the EC-1 User's Manual: Hardware.

3.2.2 CMT Control

Table 3-48 Constants Used by the CMT Driver

Constant Name	Setting Value	Description
CMT_SUCCESS	0	Constant for the function returned value. Indicates that the function has been executed successfully.
CMT_ERR	-1	Constant for the function returned value. Indicates that the function execution failed.
CMT_CH_TOTAL	4	Number of CMT channels
CMT_CH_0	0	Constant for specifying CMT channel 0.
CMT_CH_1	1	Constant for specifying CMT channel 1.
CMT_CH_2	2	Constant for specifying CMT channel 2.
CMT_CH_3	3	Constant for specifying CMT channel 3.
CMT_CKS_DIVISION_8	0	Constant for setting the clock to be input to the CMCNTn counter to PCLKD/8.
CMT_CKS_DIVISION_32	1	Constant for setting the clock to be input to the CMCNTn counter to PCLKD/32.
CMT_CKS_DIVISION_128	2	Constant for setting the clock to be input to the CMCNTn counter to PCLKD/128.
CMT_CKS_DIVISION_512	3	Constant for setting the clock to be input to the CMCNTn counter to PCLKD/512.
CMT_MODE_PERIODEC	0	Operating mode of CMT channels: Periodic event setting
CMT_MODE_ONESHOT	1	Operating mode of CMT channels: One-shot event setting
PCLKD_Hz	75000000	CMT clock setting

3.2.3 ETHER Control

Table 3-49 Constants Used by the ETHER Driver

Constant Name	Setting Value	Description
ETHER_SUCCESS	0	Function processing succeeded.
ETHER_ERR	-1	Function processing failed.

3.2.4 RSPI Control

Table 3-50 Constants Used by the RSPI Driver

Constant Name	Setting Value	Description
_RSPI_MSMODE_MASTER	0	Operating mode of RSPI channels: Master mode
_RSPI_MSMODE_SLAVE	1	Operating mode of RSPI channels: Slave mode
_RSPI_DIVISOR	0x05U	Bit rate setting in master mode
MD_STATUSBASE	0x00U	Status list base address
MD_OK	MD_STATUSBASE + 0x00U	Register setting complete
MD_SPT	MD_STATUSBASE + 0x01U	I2C communication stop
MD_NACK	MD_STATUSBASE + 0x02U	I2C NOACK
MD_BUSY1	MD_STATUSBASE + 0x03U	Busy 1 - 2
MD_BUSY2	MD_STATUSBASE + 0x04U	-
MD_ERRORBASE	0x80U	Error list base address
MD_ERROR	MD_ERRORBASE + 0x00U	error
MD_ARGERROR	MD_ERRORBASE + 0x01U	Error input error
MD_ERROR1	MD_ERRORBASE + 0x02U	Error 1 to 4
MD_ERROR2	MD_ERRORBASE + 0x03U	-
MD_ERROR3	MD_ERRORBASE + 0x04U	-
MD_ERROR4	MD_ERRORBASE + 0x05U	-
MD_ERROR5	MD_ERRORBASE + 0x06U	-
RSPI_STS_XFRCMP	0x60	Transmit / receive flag status
RSPI_STS_ERR	0x1F	Error flag status
_RSPI_MODE_SPI	0x00U	SPCR register RSPI mode select bit 0: SPI operation (4 wire type)
_RSPI_MODE_CLOCK_SYNCHRONOUS	0x01U	SPCR register RSPI mode select bit 1: Clock synchronous operation SPI operation (3 wire type)
_RSPI_FULL_DUPLEX_SYNCHRONOUS	0x00U	SPCR register Communication operation mode select bit 0: Full duplex synchronous serial communication
_RSPI_TRANSMIT_ONLY	0x02U	SPCR register Communication operation mode select bit 1: Serial communication only for transmission operation
_RSPI_MODE_FAULT_DETECT_DISABLED	0x00U	SPCR register mode fault error detection enable bit 0: Disable mode fault error detection
_RSPI_MODE_FAULT_DETECT_ENABLED	0x04U	SPCR register mode fault error detection enable bit 1: Allow mode fault error detection
_RSPI_SLAVE_MODE	0x00U	SPCR register master / slave mode select bit 0: Slave mode
_RSPI_MASTER_MODE	0x08U	SPCR register master / slave mode select bit 1: Master mode
_RSPI_ERROR_INTERRUPT_DISABLED	0x00U	SPCR register error interrupt enable bit 0: Error interrupt request generation is prohibited
_RSPI_ERROR_INTERRUPT_ENABLED	0x10U	SPCR register error interrupt enable bit 1: Enable error interrupt request generation
_RSPI_TRANSMIT_INTERRUPT_DISABLED	0x00U	SPCR register transmit buffer empty interrupt enable bit 0: Transmit buffer empty interrupt request generation disabled
_RSPI_TRANSMIT_INTERRUPT_ENABLED	0x20U	SPCR register transmit buffer empty interrupt enable bit 1: Generate transmit buffer empty interrupt request is enabled

Constant Name	Setting Value	Description
_RSPI_FUNCTION_DISABLED	0x00U	SPCR register RSPI function enable bit 0: RSPI function is invalid
_RSPI_FUNCTION_ENABLED	0x40U	SPCR register RSPI function enable bit 1: RSPI function is valid
_RSPI_RECEIVE_INTERRUPT_DISABLED	0x00U	SPCR register receive buffer full interrupt enable bit 0: Generation of receive buffer full interrupt request is
_RSPI_RECEIVE_INTERRUPT_ENABLED	0x80U	SPCR register receive buffer full interrupt enable bit 0: Generation of receive buffer full interrupt request is
_RSPI_SSL0_POLARITY_LOW	0x00U	SSLP register SSL 0 signal polarity setting bit 0: SSLy0 signal is active low
_RSPI_SSL0_POLARITY_HIGH	0x01U	SSLP register SSL 0 signal polarity setting bit 1: SSLy0 signal is active High
_RSPI_SSL1_POLARITY_LOW	0x00U	SSLP register SSL 1 signal polarity setting bit 0: SSLy1 signal is active low
_RSPI_SSL1_POLARITY_HIGH	0x02U	SSLP register SSL 1 signal polarity setting bit 1: SSLy1 signal is active High
_RSPI_SSL2_POLARITY_LOW	0x00U	SSLP register SSL 2 signal polarity setting bit 0: SSLy2 signal is active Low
_RSPI_SSL2_POLARITY_HIGH	0x04U	SSLP register SSL 2 signal polarity setting bit 1: SSLy 2 signal is active High
_RSPI_SSL3_POLARITY_LOW	0x00U	SSLP register SSL 3 signal polarity setting bit 0: SSLy3 signal is active low
_RSPI_SSL3_POLARITY_HIGH	0x08U	SSLP register SSL 3 signal polarity setting bit 1: SSLy 3 signal is active High
_RSPI_LOOPBACK_DISABLED	0x00U	SPPCR register RSPI loopback bit 0: Normal mode
_RSPI_LOOPBACK_ENABLED	0x01U	SPPCR register RSPI loopback bit 1: Loopback mode (Data is inverted and transmitted
_RSPI_LOOPBACK2_DISABLED	0x00U	SPPCR register RSPI loopback 2 bits 0: Normal mode
_RSPI_LOOPBACK2_ENABLED	0x02U	SPPCR register RSPI loopback 2 bits 1: Loopback mode (Send without inverting data)
_RSPI_OUTPUT_PIN_CMOS	0x00U	SPPCR register output terminal mode bit 0: CMOS output
_RSPI_OUTPUT_PIN_OPEN_DRAIN	0x04U	SPPCR register output terminal mode bit 1: Open drain output
_RSPI_MOSI_LEVEL_LOW	0x00U	SPPCR register MOSI idle fixed value bit 0: Output value of MOSIy pin at MOSI idle is Low
_RSPI_MOSI_LEVEL_HIGH	0x10U	SPPCR register MOSI idle fixed value bit 1: Output value of MOSIy pin at MOSI idle is High
_RSPI_MOSI_FIXING_PREV_TRANSFER	0x00U	SPPCR register MOSI Idle value fixed enable bit 0: The MOSI output value is the last data of the previous
_RSPI_MOSI_FIXING_MOIFV_BIT	0x20U	SPPCR register MOSI Idle value fixed enable bit 1: MOSI output value is set value of MOIFV bit
_RSPI_SEQUENCE_LENGTH_1 ~ 7	0x00U ~ 0x07U	SPSCR register RSPI sequence length setting bit Change reference order of SPCMD 0 ~ 7 registers to be
_RSPI_FRAMES_1 ~ 4	0x00U ~ 0x03U	SPDCR register frame number setting bit Specifying the number of frames
_RSPI_READ_SPDR_RX_BUFFER	0x00U	SPDCR register RSPI receive / transmit data select bit 0: SPDR reads receive buffer
_RSPI_READ_SPDR_TX_BUFFER	0x10U	SPDCR register RSPI receive / transmit data select bit 1: SPDR reads transmission buffer
_RSPI_ACCESS_WORD	0x00U	SPDCR register RSPI longword access / word access setting bit 0: Word access to SPDR register
_RSPI_ACCESS_LONGWORD	0x20U	SPDCR register RSPI longword access / word access setting bit 1: Longword access to SPDR register
_RSPI_RSPCK_DELAY_1 ~ 8	0x00U ~ 0x07U	SPCKD register RSPCK delay setting bit RSPCK delay setting
_RSPI_SSL_NEGATION_DELAY_1 ~ 8	0x00U ~ 0x07U	SSLND register SSL negate delay setting bit Setting SSL negation delay

Constant Name	Setting Value	Description
_RSPI_NEXT_ACCESS_DELAY_1 ~ 8	0x00U ~ 0x07U	SPND register RSPI next access delay setting bit Setting the next access delay
_RSPI_PARITY_DISABLE	0x00U	SPCR2 register Parity enable bit 0: Do not add transmit data parity bit
_RSPI_PARITY_ENABLE	0x01U	SPCR2 register Parity enable bit 1: A parity bit is added to the transmission data, and the reception data Perform a ritity check (when SPCR.TXMD = 0) Parity bit is added to transmission data, but parity check of received data is not performed (when SPCR.TXMD = 1)
_RSPI_PARITY_EVEN	0x00U	SPCR 2 register Parity mode bit 0: Send and receive with even parity
_RSPI_PARITY_ODD	0x02U	SPCR 2 register Parity mode bit 1: Transmit and receive with odd parity
_RSPI_IDLE_INTERRUPT_DISABLED	0x00U	SPCR2 register RSPI idle interrupt enable bit 0: Generation of idle interrupt request is prohibited
_RSPI_IDLE_INTERRUPT_ENABLED	0x04U	SPCR2 register RSPI idle interrupt enable bit 1: Allow generation of idle interrupt request
_RSPI_SELF_TEST_DISABLED	0x00U	SPCR 2 register Parity self-decision bit 0: Parity circuit self diagnosis function is invalid
_RSPI_SELF_TEST_ENABLED	0x08U	SPCR 2 register Parity self-decision bit 1: Parity circuit self diagnosis function is enabled
_RSPI_AUTO_STOP_DISABLED	0x00U	SPCR2 register RSPCK automatic stop function enable bit 0: RSPCK automatic stop function is invalid
_RSPI_AUTO_STOP_ENABLED	0x10U	SPCR2 register RSPCK automatic stop function enable bit 1: RSPCK automatic stop function is enabled
_RSPI_RSPCK_SAMPLING_ODD	0x0000U	SPCMD 0 to 7 Register RSPCK phase setting bit 0: Data sample at odd edge, data change at even edge
_RSPI_RSPCK_SAMPLING_EVEN	0x0001U	SPCMD 0 to 7 Register RSPCK phase setting bit 1: Data change at odd edge, data sample at even edge
_RSPI_RSPCK_POLARITY_LOW	0x0000U	SPCMD 0 - 7 Register RSPCK polarity setting bit 0: RSPCK at idle is Low
_RSPI_RSPCK_POLARITY_HIGH	0x0002U	SPCMD 0 - 7 Register RSPCK polarity setting bit 1: RSPCK at idle is High
_RSPI_BASE_BITRATE_1	0x0000U	SPCMD 0 to 7 register bit rate division setting bit 0 0: Select the base bit rate
_RSPI_BASE_BITRATE_2	0x0004U	SPCMD 0 to 7 register bit rate division setting bit 0 1: Select the base bit rate divided by 2
_RSPI_BASE_BITRATE_4	0x0008U	SPCMD 0 to 7 register bit rate division setting bit 1 0: Select the base bit rate divided by 4
_RSPI_BASE_BITRATE_8	0x000CU	SPCMD 0 to 7 register bit rate division setting bit 1 1: Select the base bit rate divided by 8
_RSPI_SIGNAL_ASSERT_SSL0 ~ 3	0x0000U ~ 0x0030U	SPCMD 0 - 7 Register SSL signal assert setting bit Setting prohibited except SSLy 0 ~ SSLy 3
_RSPI_SSL_KEEP_DISABLE	0x0000U	SPCMD 0 to 7 register SSL signal level hold bit 0: negate all SSL signals at the end of transfer
_RSPI_SSL_KEEP_ENABLE	0x0080U	SPCMD 0 to 7 register SSL signal level hold bit 1: Retain SSL signal level from transfer end to next access start
_RSPI_DATA_LENGTH_BITS_8 ~ 32	0x0400U ~ 0x0200U	SPCMD 0 - 7 Register RSPI data length setting bit
_RSPI_MSB_FIRST	0x0000U	SPCMD 0 to 7 registers RSPI LSB first bit 0: MSB first
_RSPI_LSB_FIRST	0x1000U	SPCMD 0 to 7 registers RSPI LSB first bit 1: LSB first
_RSPI_NEXT_ACCESS_DELAY_DISABLE	0x0000U	SPCMD 0 to 7 registers RSPI next access delay enable bit 0: Next access delay is 1 RSPCK + 2 SERICLK
_RSPI_NEXT_ACCESS_DELAY_ENABLE	0x2000U	SPCMD 0 to 7 registers RSPI next access delay enable bit 1: Next access delay is the setting value of RSPI next access delay register SPND)
_RSPI_NEGATION_DELAY_DISABLE	0x0000U	SPCMD 0 - 7 Register SSL negate delay setting enable bit 0: SSL negation delay is 1 RSPCK

Constant Name	Setting Value	Description
_RSPI_NEGATION_DELAY_ENABLE	0x4000U	SPCMD 0 - 7 Register SSL negate delay setting enable bit 1: SSL negation delay is the setting value of the RSPI slave select negate delay register (SSLND)
_RSPI_RSPCK_DELAY_DISABLE	0x0000U	SPCMD 0 to 7 registers RSPCK delay setting enable bit 0: RSPCK delay is 1 RSPCK
_RSPI_RSPCK_DELAY_ENABLE	0x8000U	SPCMD 0 to 7 registers RSPCK delay setting enable bit 1: The RSPCK delay is the setting value of the RSPI clock delay register (SPCKD)
_RSPI_PRIORITY_LEVEL0 ~ 15	0x00000000UL ~ 0x0000000FUL	PRLn register Interrupt priority level storage bit Interrupt priority levels are 0 for the highest and 15 for the lowest priority.

3.2.5 SCIFA_UART Control

Table 3-51 Constants Used by the SCIFA_UART Driver

Constant Name	Setting Value	Description
SCIFA_HVA_WRITE_DATA	0u	Initializing the interrupt address register
SCIFA_UART_SUCCESS	0	Success
SCIFA_UART_ERR	-1	Error
SCIFA_UART_ERR_RECEIVE	-2	Reception error
SCIFA_UART_CH_TOTAL	5	Number of SCIFA channels
SCIFA_UART_CH_0	0	Constant for specifying SCIFA channel 0.
SCIFA_UART_CH_1	1	Constant for specifying SCIFA channel 1.
SCIFA_UART_CH_2	2	Constant for specifying SCIFA channel 2.
SCIFA_UART_CH_3	3	Constant for specifying SCIFA channel 3.
SCIFA_UART_CH_4	4	Constant for specifying SCIFA channel 4.
SCIFA_UART_MODE_R	1	Constant for using SCIFA in reception mode Used as the argument for the SCIFA channel initialization function and the SCIFA channel open function.
SCIFA_UART_MODE_W	2	Constant for using SCIFA in transmission mode Used as the argument for the SCIFA channel initialization function and the SCIFA channel open function.
SCIFA_UART_MODE_RW	3	Constant for using SCIFA in transmission/reception mode Used as the argument for the SCIFA channel initialization function and the SCIFA channel open function.
SCIFA_UART_CKS_DIVISION_1	0	Constant for setting the SERICLK clock as the clock source of the baud rate generator of the SCIFA. Used as the argument for the SCIFA channel initialization function.
SCIFA_UART_CKS_DIVISION_4	1	Constant for setting the SERICLK/4 clock as the clock source of the baud rate generator of the SCIFA. Used as the argument for the SCIFA channel initialization function.
SCIFA_UART_CKS_DIVISION_16	2	Constant for setting the ERICLK/16 clock as the clock source of the baud rate generator of the SCIFA. Used as the argument for the SCIFA channel initialization function.
SCIFA_UART_CKS_DIVISION_64	3	Constant for setting the ERICLK/64 clock as the clock source of the baud rate generator of the SCIFA. Used as the argument for the SCIFA channel initialization function.

3.2.6 Serial Flash ROM Control

Table 3-52 Constants Used by the Serial Flash ROM Driver

Constant Name	Setting Value	Description
SFLASH_BASE_ADDRESS	0x10000000	Serial flash ROM base address
SFLASH_CHANNEL_MAX	1	Number of channels
SFLASH_DEVICE_SIZE	crr_sflash_info.device_size	Device information structure-device_size
SFLASH_ERASE_SIZE	crr_sflash_info.erase_size	Device information structure-erase_size
SFLASH_PROGRAM_SIZE	crr_sflash_info.page_size	Device information structure-page_size
SFLASH_ID	crr_sflash_info.id	Device information structure-id

3.2.7 USB Function Control

Table 3-53 Constants Used by the USB Function Driver

Constant Name	Setting Value	Description
USB_DMA_USE_PP	1	DMA transfer used
USB_DMA_NOT_USE_PP	0	DMA transfer not used
USB_DMA_PP	USB_DMA_USE_PP	DMA transfer setting
USB_LPWR_USE_PP	1	Low power mode used
USB_LPWR_NOT_USE_PP	0	Low power mode not used
USB_CPU_LPW_PP	USB_LPWR_USE_PP	Low power mode setting
USB_DEBUG_ON_PP	1	Debugging information output enabled
USB_DEBUG_OFF_PP	0	Debugging information output disabled
USB_DEBUG_OUTPUT_PP	USB_DEBUG_OFF_PP	Debugging information output function setting
USB_OK	USB_ER_t(0)	Success
USB_ERROR	USB_ER_t(-1L)	Error
USB_QOVR	USB_ER_t(-43L)	The specified pipe is processing data transfer.
USB_TRUE	1	Remote wakeup enable flag: Enabled
USB_FALSE	0	Remote wakeup enable flag: Disabled
USB_NULL	0	NULL
USB_CS_SQER	0x0006u	Sequence error
USB_CS_WRND	0x0005u	Ctrl write no data status stage
USB_CS_WRSS	0x0004u	Ctrl write status stage
USB_CS_WRDS	0x0003u	Ctrl write data stage
USB_CS_RDSS	0x0002u	Ctrl read status stage
USB_CS_RDDS	0x0001u	Ctrl read data stage
USB_CS_IDST	0x0000u	Idle or setup stage
USB_CUSE	0u	The CPU accesses CFIFO.
USB_D0DMA	2u	The DMA accesses D0FIFO (cycle steal mode).
USB_D0DMA_C	6u	The DMA accesses D0FIFOBn (32-byte continuous access mode).
USB_D1DMA_C	7u	The DMA accesses D1FIFOBn (32-byte continuous access mode).
USB_FIFOERROR	0xFFFFu	An FIFO access error occurred.
USB_WRITEEND	0x0000u	Data write finished (no continued data or packet transmission of data length 0)
USB_WRITESHRT	0x0001u	Data write finished (short packet data write)
USB_WRITING	0x0002u	Data being written (there is continued data)
USB_PDTBLEND	0xFFFFu	End of the pipe information table
USB_CTRL_END	0u	Ends the status stage normally.
USB_DATA_NONE	1u	When data transmission was ended normally

Constant Name	Setting Value	Description
USB_DATA_OK	3u	When data reception was ended normally
USB_DATA_SHT	4u	When the data length was less than the specified length although data reception was ended normally
USB_DATA_OVR	5u	When the received data exceeded the specified size
USB_DATA_STALL	6u	When a STALL response or MaxPacketSiz error was detected
USB_DATA_ERR	7u	Returns STALL to the host in the status stage.
USB_DATA_STOP	8u	When data transfer was ended forcibly
USB_DO_REMOTEWAKEUP	0x0155u	Makes a remote wakeup execution request to the PCD.
USB_DO_STALL	0x0164u	Makes a STALL response execution request to the PCD.
USB_PCDC_USE_PIPE_IN	USB_PIPE1	Bulk IN transfer pipe number
USB_PCDC_USE_PIPE_OUT	USB_PIPE2	Bulk OUT transfer pipe number
USB_PCDC_USE_PIPE_STATUS	USB_PIPE6	Interrupt IN transfer pipe number
USB_HSCONNECT	0x00C0u	Operating device speed: High-speed
USB_FSCONNECT	0x0080u	Operating device speed: Full-speed
USB_NOCONNECT	0x0000u	Operating device speed: No connect
USB_PIPE0	0x0000u	Used pipe (USB_PIPE1 to USB_PIPE9)
USB_PIPE1	0x0001u	-
USB_PIPE2	0x0002u	-
USB_PIPE3	0x0003u	-
USB_PIPE4	0x0004u	-
USB_PIPE5	0x0005u	-
USB_PIPE6	0x0006u	-
USB_PIPE7	0x0007u	-
USB_PIPE8	0x0008u	-
USB_PIPE9	0x0009u	-
USB_BULK	0x4000u	Pipe transfer type (USB_BULK/USB_INT/USB_ISO)
USB_INT	0x8000u	-
USB_ISO	0xC000u	-
USB_BFREOFF	0x0000u	BRDY interrupt operation specification
USB_DBLBON	0x0200u	Double buffer mode
USB_DBLBOFF	0x0000u	-
USB_CNTMDON	0x0100u	Continuous transfer mode
USB_CNTMDOFF	0x0000u	-
USB_SHTNAKON	0x0080u	SHTNAK operation specification
USB_SHTNAKOFF	0x0000u	-
USB_DIR_P_OUT	0x0000u	Transfer direction
USB_DIR_P_IN	0x0010u	-
USB_EP1	0x0001u	Endpoint number (EP1 to EP15) for pipes
USB_EP2	0x0002u	-
USB_EP3	0x0003u	-
USB_EP4	0x0004u	-
USB_EP5	0x0005u	-
USB_EP6	0x0006u	-
USB_EP7	0x0007u	-
USB_EP8	0x0008u	-

Constant Name	Setting Value	Description
USB_EP9	0x0009u	-
USB_EP10	0x000Au	-
USB_EP11	0x000Bu	-
USB_EP12	0x000Cu	-
USB_EP13	0x000Du	-
USB_EP14	0x000Eu	-
USB_EP15	0x000Fu	-

3.2.8 WDTA Control

Table 3-54 Constants Used by the WDTA Driver

Constant Name	Setting Value	Description
WDT_STAT_REFRESH_ERR_MASK	(0x8000)	This constant shows definition to acquire the refresh error flag from the WDT status.
WDT_STAT_UNDERFLOW_ERR_MASK	(0x4000)	This constant shows definition to acquire the underflow flag from the WDT status.
WDT_STAT_ERROR_MASK	(0xC000)	This constant shows definition to acquire the refresh error flag and the underflow flag from the WDT status.
WDT_STAT_COUNTER_MASK	(0x3FFF)	This constant shows definition to acquire the counter value from the WDT status.
WDT_CFG_PARAM_CHECKING_ENABLE	(1)	This constant shows whether the parameter check using the WDTA's API function is enabled (1) or disabled (0).

3.2.9 IWDTA Control

Table 3-55 Constants Used by the IWDTA Driver

Constant Name	Setting Value	Description
IWDT_STAT_REFRESH_ERR_MASK	(0x8000)	This constant shows definition to acquire the refresh error flag from the IWDT status.
IWDT_STAT_UNDERFLOW_ERR_MASK	(0x4000)	This constant shows definition to acquire the underflow flag from the IWDT status.
IWDT_STAT_ERROR_MASK	(0xC000)	This constant shows definition to acquire the refresh error flag and the underflow flag from the IWDT status.
IWDT_STAT_COUNTER_MASK	(0x3FFF)	This constant shows definition to acquire the counter value from the IWDT status.
IWDT_CFG_PARAM_CHECKING_ENABLE	(1)	This constant shows whether the parameter check using the IWDTA's API function is enabled (1) or disabled (0).

3.2.10 RIIC Control

Table 3-56 lists constants used by the RIIC driver, and Table 3-57 lists error codes used by the RIIC driver.
Table 3-58 and Table 3-59 list constants that can be configured at the time of compiling.

Table 3-56 Constants Used by the RIIC Driver

Constant Name	Setting Value	Description
RIIC_NO_INIT Note 1	0	Uninitialized state
RIIC_IDLE Note 1	1	Idle state
RIIC_FINISH Note 1	2	Idle state
RIIC_NACK Note 1	3	Idle state
RIIC_COMMUNICATION Note 1	4	Master transmission/reception state and slave transmission/reception state
RIIC_AL Note 1	5	Arbitration lost detection state
RIIC_ERROR Note 1	6	Error state
RIIC_GEN_START_CON Note 2	(uint8_t)(0x01)	Generation of start condition
RIIC_GEN_STOP_CON Note 2	(uint8_t)(0x02)	Generation of stop condition
RIIC_GEN_RESTART_CON Note 2	(uint8_t)(0x04)	Generation of restart condition
RIIC_GEN_SDA_HI_Z Note 2	(uint8_t)(0x08)	Hi-Z output from the SDA pin
RIIC_GEN_SCL_ONESHOT Note 2	(uint8_t)(0x10)	One-shot output of SCL clock
RIIC_GEN_RESET Note 2	(uint8_t)(0x20)	RIIC module reset
FIT_NO_PTR	(void *)0	NULL pointer defined by FIT

Note 1: Used as a value of riic_ch_dev_status_t type

Note 2: Used as an output pattern of R_RIIC_Control()

Table 3-57 Error Codes Used by the RIIC Driver

Constant Name	Setting Value	Description
RIIC_SUCCESS	0U	The function has been successfully called.
RIIC_ERR_LOCK_FUNC	1U	RIIC is used by another module.
RIIC_ERR_INVALID_CHAN	2U	A non-existent channel was specified.
RIIC_ERR_INVALID_ARG	3U	An invalid argument was set.
RIIC_ERR_NO_INIT	4U	In the uninitialized state
RIIC_ERR_BUS_BUSY	5U	In the bus busy state
RIIC_ERR_AL	6U	A function was called in the arbitration lost detection state.
RIIC_ERR_OTHER	7U	Other errors

The r_riic_ec1_config.h file is used to set configuration options of this module.

Option names and set values are described below.

Table 3-58 Settings for Compiling (1/2)

Definition	Set Value
RIIC_CFG_PARAM_CHECKING_ENABLE Note: Default value = 1	Whether to include the parameter check processing in the code is selectable. When 0 is selected, the parameter check processing can be deleted from the code and the code size can be reduced. When 0 is set, the parameter check processing is deleted from the code. When 1 is set, the parameter check processing is included in the code.
RIIC_CFG_PCLK_Hz Note: Default value = 75000000	Set the frequency of the clock (PCLK) supplied to the RIIC1 module. The setting values of the bit rate register and the internal reference clock select bits are calculated from the set values of “RIIC_CFG_CH0_kBPS” and “RIIC_CFG_PCLK_Hz.”
RIIC_CFG_CH_NUMBER Note: Default value = 1	Set the channel to be used for RIIC communication.
RIIC_CFG_CH_kBPS Note: Default value = 400	The transmission rate of RIIC1 can be set. The setting values of the bit rate register and the internal reference clock select bits are calculated from the set values of “RIIC_CFG_CH_kBPS” and “RIIC_CFG_PCLK_Hz.” Set a value not more than 400.
RIIC_SCL_100K_UP_TIME Note: Default value = 1000E-9	Set the SCL rising time [s] when the transmission rate of RIIC1 is 1 to 100 [KBPS]. (Specify a double-type value.)
RIIC_SCL_100K_DOWN_TIME Note: Default value = 300E-9	Set the SCL falling time [s] when the transmission rate of RIIC1 is 1 to 100 [KBPS]. (Specify a double-type value.)
RIIC_SCL_400K_UP_TIME Note: Default value = 175E-9	Set the SCL rising time [s] when the transmission rate of RIIC1 is 101 to 400 [KBPS]. (Specify a double-type value.)
RIIC_SCL_400K_DOWN_TIME Note: Default value = 175E-9	Set the SCL falling time [s] when the transmission rate of RIIC1 is 101 to 400 [KBPS]. (Specify a double-type value.)
RIIC_CFG_CH_DIGITAL_FILTER Note: Default value = 2	The number of noise filters is selectable. When 0 is set, noise filters are deactivated. When a value of 1 to 4 is set, the set values of the noise filter number select bits and the digital noise filter circuit enable bit are selected so that the selected filters are enabled.
RIIC_CFG_CH_SCL Note: Default value = 1	The SCL output pin of RIIC1 is selectable. The processing to use the selected pin as SCL pin is included in the code. When 0 is set, the SCL pin setting processing is deleted from the code. When 1 is set, PC6 is set as SCL pin.
RIIC_CFG_CH_SDA Note: Default value = 1	The SDA output pin of RIIC1 is selectable. The processing to use the selected pin as SDA pin is included in the code. When 0 is set, the SDA pin setting processing is deleted from the code. When 1 is set, PC7 is set as SDA pin.

RIIC_CFG_CH_MASTER_MODE Note: Default value = 1	The master arbitration lost detection function can be enabled or disabled. When using this function in the multi-master, set this value to 1 (enabled). When 0 is set, master arbitration lost detection is disabled. When 1 is set, master arbitration lost detection is enabled.
RIIC_CFG_CH_SLV_ADDR0_FORMAT Note 1 RIIC_CFG_CH_SLV_ADDR1_FORMAT Note 2 RIIC_CFG_CH_SLV_ADDR2_FORMAT Note 2 Note 1: Default value = 1 Note 2: Default value = 0	The slave address format can be selected from 7 bits or 10 bits. When 0 is set, no slave address is set. When 1 is set, 7-bit address format is selected. When 2 is set, 10-bit address format is selected.
RIIC_CFG_CH_SLV_ADDR0 Note 1 RIIC_CFG_CH_SLV_ADDR1 Note 2 RIIC_CFG_CH_SLV_ADDR2 Note 2 Note 1: Default value = 0x0025 Note 2: Default value = 0x0000	The slave address can be set. The settable range varies with the set value of “RIIC_CFG_CH_SLV_ADDRi_FORMAT.” When “RIIC_CFG_CH_SLV_ADDRi_FORMAT” = 0, the set value is disabled. When this value = 1, the lower 7 bits of the set value are enabled. When this value = 2, the lower 10 bits of the set value are enabled.

Table 3-59 Settings for Compiling (2/2)

Definition	Set Value
RIIC_CFG_CH_SLV_GCA_ENABLE Note: Default value = 0	The general code address can be enabled or disabled. When 0 is set, the general code address is disabled. When 1 is set, the general code address is enabled.
RIIC_CFG_CH_INT_PRIORITY Note: Default value = 1	The priority level of communication error/event interrupt (EEI), receive data full interrupt (RXI), transmit data empty interrupt (TXI), and transmission end interrupt (TEI) is selectable. Set a value of 1 to 15.
RIIC_CFG_BUS_CHECK_COUNTER Note: Default value = 1000	The timeout counter (number of bus check times) for the bus check processing of the API function of RIIC can be set. Set a value not more than “0xFFFFFFFF.” The bus check processing is performed; ・before start condition is created, ・after stop condition is detected, or ・each condition and SCL one-shot pulse using the RIIC control function (R_RIIC_Control function) are created. In the bus check processing, the timeout counter is decremented in the bus busy state until the bus busy is released. When the counter becomes 0, it is decided as timeout and an error (Busy) is returned in the return value. Note: Because this counter is used to prevent deadlock due to a bus lock, set a value longer than the time where the remote device holds the SCL pin at low level. Timeout period (ns) ≒ (1 / ICLK (Hz)) * counter value * 10

3.2.11 USB Host Control

USB-BASIC-F/W functions can be set by rewriting the common user definition information file (r_usb_basic_config.h).

Change the following items according to the system.

Table 3-60 Constants Used by the Host Control Driver (HCD)

Constant	Set Value	Description
USB_MAXDEVADDR	12u	Sets the maximum value of the device address (maximum number of connectable devices). Setting a large value increases the size of memory used.
USB_IDMAX	11u	Sets the maximum value of the task ID of the scheduler. Setting a large value increases the size of memory used.
USB_PRIMAX	8u	Sets the maximum value of the priority of the scheduler. Setting a large value increases the size of memory used.
USB_BLKMAX	21u	Sets the maximum number of memory pool of the scheduler. Setting a large value increases the size of memory used.
USB_TABLEMAX	USB_BLKMAX	Sets the maximum number of message pool of the scheduler. Setting a large value increases the size of memory used.
USB_MAXPIPE	32u	Sets the maximum number of pipes. One pipe is used for each endpoint of connected devices. Setting a large value increases the size of memory used.
USB_HOST_COMPLIANCE_MODE	-	Sets whether to respond to the compliance test of the USB Embedded Host. To respond to the compliance test, enable this macro.
USB_COMPLIANCE_DISP(data1, data2)	usb_cstd_DummyFunction(dat	Register the function to display information on a display device (such as LCD) to respond to the compliance test.
USB_OVERCURRENT(rootport)	usb_cstd_DummyFunction(rootport)	Register the function to be called when overcurrent is detected.
USB_DEBUG_OUTPUT	-	Sets debug information output.
USB_PRINTF0(FORM) to USB_PRINTF8(FORM,x1,x2,x3,x4,x5,x6,x7,x8)	printf(FORM) to printf (FORM,x1,x2,x3,x4,x5,x6,x7,x	This macro outputs debug information to a UART and a display device. A serial driver and a display device driver are required. Register a function according to the user system.

Table 3-61 Setting the EHCI User Definition File (r_usb_hEhciDefUsr.h)

Constant	Set Value	Description
USB_EHCI_PFL_SIZE	256	Specify the size of the EHCI periodic frame list from values 256, 512, and 1024. Specifying this size changes the periodic transfer scheduling width.
USB_EHCI_NUM_QH	16	Specify the maximum size of the EHCI queue head data structure memory. A size of the number of endpoint pipes for control transfer, bulk transfer, and interrupt transfer is required for the queue head data structure. Set a value according to the user system.
USB_EHCI_NUM_QTD	256	Specify the maximum size of the EHCI qTD (device status register queue element transfer descriptor) data structure memory. The qTD descriptor is used to manage control transfer, bulk transfer, and interrupt transfer in conjunction with the queue head. Data of up to 20480 bytes can be transferred with a single qTD descriptor. For control transfer, a qTD descriptor is required for each SETUP stage, DATA stage, and STATUS stage. Set a value according to the user system.
USB_EHCI_NUM_ITD	4	Specify the maximum size of the EHCI iTD (high-speed isochronous transfer descriptor) data structure memory. A size of the number of endpoint pipes for high-speed isochronous transfer is required for the iTD data structure. Set a value according to the user system.
USB_EHCI_NUM_SITD	4	Specify the maximum size of the EHCI siTD (split transaction isochronous transfer descriptor) data structure memory. A size of the number of endpoint pipes for split transaction isochronous transfer is required for the siTD data structure. Set a value according to the user system.
USB_EHCI_ITD_DATA_SIZE	512	Specify the maximum data transfer size of the EHCI iTD data structure. This transfer size specifies the maximum transfer size for one transaction of high-speed isochronous transfer. A data size of up to 1024 can be set. Set a value according to the user system.
USB_EHCI_TIMEOUT	3000	Specify the EHCI timeout period [msec].

Table 3-62 Setting the OHCI User Definition File (r_usb_hOhciDefUsr.h)

Constant	Set Value	Description
USB_OHCI_NUM_ENDPOINT	16	Specify the maximum size of the OHCI endpoint data structure memory. A size of the number of endpoint pipes for control transfer, bulk transfer, interrupt transfer, and isochronous transfer is required for the OHCI endpoint data structure. Set a value according to the user system.
USB_OHCI_NUM_ED	64	Specify the maximum size of the OHCI endpoint descriptor data structure memory. Thirty-one OHCI endpoint descriptor data structures are required for endpoint pipes for control transfer, bulk transfer, interrupt transfer, and isochronous transfer and for scheduling interrupt transfer. Set a value according to the user system.
USB_OHCI_NUM_TD	256	Specify the maximum size of the OHCI transfer descriptor data structure memory. The OHCI transfer descriptor data structure is used as a descriptor to manage control transfer, bulk transfer, interrupt transfer, and isochronous transfer. Data of up to 8192 bytes can be transferred with a single transfer descriptor. Set a value according to the user system.
USB_OHCI_ISO_MAXDEVICE	4	Specify the maximum number of OHCI isochronous devices. Set a value according to the user system.
USB_OHCI_ISO_MAX_PACKET_SIZE	256	Specify the maximum data transfer size of OHCI isochronous devices. This transfer size specifies the maximum transfer size for one transaction of OHCI isochronous transfer. A data size of up to 1023 can be set. Set a value according to the user system.
USB_OHCI_ISO_MAX_FRAME	8	Specify the maximum number of OHCI isochronous transfer frames. This number of frames specifies the maximum number of frames for OHCI isochronous transfer. A power of 2 can be set. The settable maximum number of frames = 8. Set a value according to the user system.
USB_OHCI_TIMEOUT	3000	Specify the OHCI timeout period [msec].

3.3 Functions

The table below lists the functions the drivers provide.

Table 3-63 List of CAN Driver Functions

Function Name	Description	Header
R_CAN_Open	Starts up the CAN module	r_can_api.h
R_CAN_Close	Stops the CAN module	r_can_api.h
R_CAN_GlobalControl	Makes a transition between the global modes	r_can_api.h
R_CAN_ChannelControl	Makes a transition between the channel modes	r_can_api.h
R_CAN_SetBitrate	Sets the transfer rates	r_can_api.h
R_CAN_UseBufferEntry	Registers the information of the buffers for use in transmission and reception	r_can_api.h
R_CAN_SetRxFifoBuffer	Enables the reception FIFO buffer	r_can_api.h
R_CAN_SetFifoBuffer	Enables the transmission/reception FIFO buffer	r_can_api.h
R_CAN_ReleaseFifoBuffer	Releases the transmission/reception FIFO buffer	r_can_api.h
R_CAN_ReleaseRxFifoBuffer	Releases the reception FIFO buffer	r_can_api.h
R_CAN_ReleaseBuffer	Releases the transmission buffer or the reception buffer	r_can_api.h
R_CAN_GetTxBufferStatus	Reads the state of the transmission buffer	r_can_api.h
R_CAN_WriteBuffer	Writes messages to be transmitted to the transmission buffer	r_can_api.h
R_CAN_GetFifoStatus	Reads the state of the transmission/reception FIFO buffer	r_can_api.h
R_CAN_WriteFifo	Writes messages to be transmitted to the transmission/reception FIFO buffer	r_can_api.h
R_CAN_Tx	Starts transmission	r_can_api.h
R_CAN_RxSet	Makes settings for reception	r_can_api.h
R_CAN_ReadBuff	Reads received messages from the reception buffer	r_can_api.h
R_CAN_ReadRxFifo	Reads received messages from the reception FIFO buffer	r_can_api.h
R_CAN_ReadFifo	Reads received messages from the transmission/reception FIFO buffer	r_can_api.h
R_CAN_GetFifoMessageNum	Gets the number of unread messages in the transmission/reception FIFO buffer	r_can_api.h
R_CAN_GetRxFifoMessageNum	Gets the number of unread messages in the reception FIFO buffer	r_can_api.h
R_CAN_SetCommTestMode	Makes settings for transfer tests	r_can_api.h
R_CAN_ResetTestMode	The module is released from the test mode and enters the channel transfer mode	r_can_api.h
R_CAN_SetInterruptHandler	Registers the interrupt handler	r_can_api.h
R_CAN_SetInterruptEnableDisable	Enables or disables the CAN module interrupt vectors	r_can_api.h
R_CAN_GetInterruptSource	Gets the interrupt source	r_can_api.h
R_CAN_ClearInterruptSource	Clears the interrupt source	r_can_api.h

Table 3-64 List of CMT Driver Functions

Function Name	Description	Header
R_CMT_Init	Initializes CMT channels	r_cmt.h
R_CMT_CreatePeriodic	Makes periodic event settings	r_cmt.h
R_CMT_CreateOneShot	Makes one-shot event settings	r_cmt.h
R_CMT_Stop	Stops CMT operation	r_cmt.h
userdef_cmt_init	Initializes the CMT	r_cmt.h
userdef_cmt_create	Makes CMT interval settings	r_cmt.h
userdef_cmt_stop	Stops CMT operation	r_cmt.h
userdef_cmt_isr_cmi	CMI interrupt handler	r_cmt.h
cmt0_isr	CMI0 interrupt handler	-
cmt1_isr	CMI1 interrupt handler	-
cmt2_isr	CMI2 interrupt handler	-
cmt3_isr	CMI3 interrupt handler	-

Table 3-65 List of ETHER Driver Functions

Function Name	Description	Header
EtherCAT_init	Initializes EtherCAT®	r_ether.h
R_ETHER_IRQ_Init	Initializes Ether interrupt requests	r_ether.h
ether_irq_init	Initializes Ether interrupt requests	-
ecat_sync0_isr	EtherCAT SYNC signal output pin 0 interrupt handler	-
ecat_sync1_isr	EtherCAT SYNC signal output pin 1 interrupt handler	-
ecat_isr	EtherCAT interrupt handler	-

Table 3-66 List of RSPI Driver Functions

M used in RSPIm represents the channel number.

Function Name	Description	Header
R_RSPIm_Create	Initializes RSPI control.	r_rspi.h
R_RSPIm_Pin_Init	RSPI port setting initialization process.	r_rspi.h
R_RSPIm_Start	Start RSPI communication.	r_rspi.h
R_RSPIm_Stop	Stops RSPI communication.	r_rspi.h
R_RSPI_GetState	Gets the RSPI communication status.	r_rspi.h
R_RSPIm_ClearState	RSPI communication status initialization process.	r_rspi.h
R_RSPIm_Send_Receive	Starts RSPI transmission and reception.	r_rspi.h
r_rspim_callback_transmitend	Processing when transmission ends.	r_rspi.h
r_rspim_callback_receiveend	Processing of a reception buffer full interrupt.	r_rspi.h
r_rspim_callback_error	Processing of an RSPI error interrupt.	r_rspi.h
r_rspim_transmit_interrupt	Processing of a transmission buffer empty interrupt.	r_rspi.h
r_rspim_receive_interrupt	Processing of a reception buffer full interrupt.	r_rspi.h

Table 3-67 List of SCFIA_UART Driver Functions

Function Name	Description	Header
R_SCIFA_SetInterruptHandler	Registers the SCIFA interrupt handler	r_scifa_api.h
R_SCIFA_EnableItrReg	Enables SCIFA reception interrupts	r_scifa_api.h
R_SCIFA_ResetItrReg	Disables SCIFA reception interrupts	r_scifa_api.h
R_SCIFA_UART_Init	Initializes the SCIFA	r_scifa_uart.h
R_SCIFA_UART_Open	Starts the SCIFA	r_scifa_uart.h
R_SCIFA_UART_Receive	Performs SCIFA data reception	r_scifa_uart.h
R_SCIFA_UART_Send	Performs SCIFA data transmission	r_scifa_uart.h
userdef_scifa0_uart_init	Initializes SCIFA channel 0 UART mode	r_scifa_uart.h
userdef_scifa0_uart_open	Starts SCIFA channel 0 UART mode	r_scifa_uart.h
userdef_scifa0_uart_receive	Receives data in SCIFA channel 0 UART mode	r_scifa_uart.h
userdef_scifa0_uart_send	Transmits data in SCIFA channel 0 UART mode	r_scifa_uart.h
_write	Outputs character strings	siorw.c
_read	Inputs character strings	siorw.c
IoInitScifa0	Initializes input/output (initializes SCIFA channel 0)	siochar.c
IoGetchar	Gets characters	siochar.c
IoPutchar	Outputs characters	siochar.c

Table 3-68 List of Serial Flash ROM Driver Functions

Function Name	Description	Header
R_SFLASH_init	Initializes serial flash ROM control	r_sflash.h
R_SFLASH_program	Writes data to serial flash ROM	r_sflash.h
R_SFLASH_erase	Erases data from serial flash ROM	r_sflash.h

Table 3-69 List of USB Function Driver Functions

Function Name	Description	Header
R_USB_Open	Initializes the USB module	r_usb_basic_if.h
R_usb_pstd_TransferStart	Requests data transfer	r_usb_basic_if.h
R_usb_pstd_TransferEnd	Requests data transfer forced end	r_usb_basic_if.h
R_usb_pstd_ChangeDeviceState	Makes a transition between USB device states	r_usb_basic_if.h
R_usb_pstd_DriverRegistration	Registers PDCD information	r_usb_basic_if.h
R_usb_cstd_SetBuf	Sets the PID bits for the specified pipe to BUF	r_usb_basic_if.h
R_usb_pstd_SetPipeStall	Sets the PID bits for the specified pipe to STALL	r_usb_basic_if.h
R_usb_pstd_ControlRead	Requests data transfer for control IN transfer	r_usb_basic_if.h
R_usb_pstd_ControlWrite	Requests data transfer for control OUT transfer	r_usb_basic_if.h
R_usb_pstd_ControlEnd	Requests control transfer end	r_usb_basic_if.h
R_usb_pstd_poll	USB interrupt processing	r_usb_basic_if.h
R_usb_pcdc_SendData	USB transmission processing	r_usb_pcdc_if.h
R_usb_pcdc_ReceiveData	USB reception processing	r_usb_pcdc_if.h
R_usb_pcdc_SerialStateNotification	Sends class notification “Serial State”	r_usb_pcdc_if.h
R_usb_pcdc_ctrltrans	Control transfer for CDC	r_usb_pcdc_if.h

Table 3-70 WDTA Driver Functions

Function	Description	Header
R_WDT_Open	Opens the WDT.	r_wdt_if.h
R_WDT_Control	Controls the WDT.	r_wdt_if.h

Table 3-71 IWDTA Driver Functions

Function	Description	Header
R_IWDT_Open	Opens the IWDT.	r_iwdt_if.h
R_IWDT_Control	Controls the IWDT.	r_iwdt_if.h

Table 3-72 RIIC Driver Functions

Function	Description	Header
R_RIIC_Open	Activates the RIIC module.	r_riic_ec1_if.h
R_RIIC_MasterSend	Starts RIIC master transmission.	r_riic_ec1_if.h
R_RIIC_MasterReceive	Starts RIIC master reception.	r_riic_ec1_if.h
R_RIIC_SlaveTransfer	Makes a transition to RIIC slave transmission/reception state.	r_riic_ec1_if.h
R_RIIC_GetStatus	Checks the RIIC module status.	r_riic_ec1_if.h
R_RIIC_Control	Performs RIIC control processing.	r_riic_ec1_if.h
R_RIIC_Close	Deactivates the RIIC module.	r_riic_ec1_if.h
R_RIIC_GetVersion	Acquires the RIIC version.	r_riic_ec1_if.h

Table 3-73 USB Host USB-BASIC-F/W API Functions

Function	Description	Header
R_usb_hstd_TransferStart	Requests execution of data transfer.	r_usb_basic_if.h
R_usb_hstd_TransferEnd	Requests forcible termination of data transfer.	r_usb_basic_if.h
R_usb_hstd_DriverRegistration	Registers the HDCD.	r_usb_basic_if.h
R_usb_hstd_ReturnEnuMGR	Reports completion of class check.	r_usb_basic_if.h
R_usb_hstd_ChangeDeviceState	Requests status change of connected devices.	r_usb_basic_if.h
R_usb_hstd_SetPipe	Sets pipe information.	r_usb_basic_if.h
R_usb_hstd_GetPipeID	Acquires the pipe number.	r_usb_basic_if.h
R_usb_hstd_ClearPipe	Clears pipe information.	r_usb_basic_if.h
R_usb_hstd_MgrOpen	Activates the MGR task.	r_usb_basic_if.h
R_usb_hstd_MgrTask	MGR task	r_usb_basic_if.h
R_usb_hhub_Registration	Registers the HUB Class Driver (HUBCD).	r_usb_basic_if.h
R_usb_hhub_Task	HUB task	r_usb_basic_if.h

Table 3-74 USB Host non-OS Scheduler API Functions

Function	Description	Header
R_usb_cstd_SndMsg	Sends a processing request to the priority table.	r_usb_basic_if.h
R_usb_cstd_RecMsg	Checks whether the priority table includes a processing request.	r_usb_basic_if.h
R_usb_cstd_PgetBlk	Ensures an area to store processing requests.	r_usb_basic_if.h
R_usb_cstd_RelBlk	Ensures the area that stores processing requests.	r_usb_basic_if.h
R_usb_cstd_Scheduler	Manages processing requests from each task.	r_usb_basic_if.h
R_usb_cstd_SetTaskPri	Sets task priority.	r_usb_basic_if.h
R_usb_cstd_CheckSchedule	Checks whether registered tasks have processing requests.	r_usb_basic_if.h

Table 3-75 USB Host HMSCD API Functions

Function	Description	Header
R_usb_hmsc_Task	HMSCD task	r_usb_hmsc_if.h
R_usb_hmsc_driver_start	Activates the HMSC driver.	r_usb_hmsc_if.h
R_usb_hmsc_ClassCheck	Performs descriptor check processing.	r_usb_hmsc_if.h
R_usb_hmsc_GetDevSts	Sends a response of the HMSCD operating status.	r_usb_hmsc_if.h
R_usb_hmsc_Read10	Issues the READ10 command.	r_usb_hmsc_if.h
R_usb_hmsc_Write10	Issues the WRITE10 command.	r_usb_hmsc_if.h
R_usb_hmsc_GetMaxUnit	Issues a GetMaxLUN request.	r_usb_hmsc_if.h
R_usb_hmsc_MassStorageReset	Issues a MassStorageReset request.	r_usb_hmsc_if.h
R_usb_hmsc_alloc_drvno	Allocates drive numbers.	r_usb_hmsc_if.h
R_usb_hmsc_free_drvno	Releases drive numbers.	r_usb_hmsc_if.h
R_usb_hmsc_ref_drvno	References drive numbers.	r_usb_hmsc_if.h

Table 3-76 USB Host HMSDD API Functions

Function	Description	Header
R_usb_hmsc_StrgDriveTask	Master storage driver task	r_usb_hmsc_if.h
R_usb_hmsc_StrgDriveSearch	Acquires drive information.	r_usb_hmsc_if.h
R_usb_hmsc_StrgDriveOpen	Opens the drive.	r_usb_hmsc_if.h
R_usb_hmsc_StrgDriveClose	Closes the drive.	r_usb_hmsc_if.h
R_usb_hmsc_StrgReadSector	Reads sectors.	r_usb_hmsc_if.h
R_usb_hmsc_StrgWriteSector	Writes sectors.	r_usb_hmsc_if.h
R_usb_hmsc_StrgCheckEnd	Checks whether data read/write has been completed.	r_usb_hmsc_if.h
R_usb_hmsc_StrgUserCommand	Issues the storage command.	r_usb_hmsc_if.h

Table 3-77 USB Host FSI API Functions

Function	Description	Header
disk_status	Acquires the device status.	diskio.h
disk_initialize	Initializes the device.	diskio.h
disk_read	Reads data.	diskio.h
disk_write	Writes data.	diskio.h
disk_ioctl	Controls other devices.	diskio.h
get_fattime	Acquires date and time information.	ff.h

3.4 CAN Control

This LSI provides drivers for performing communication using the CAN module (RSCAN).

3.4.1 Starting up the CAN Module

R_CAN_Open

(1) **Synopsis**

This is the function used first when using the CAN module.

(2) **C language format**

void R_CAN_Open(uint32_t ch, uint32_t frequency);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	uint32_t	frequency	Transfer rate

(4) **Description**

This function makes initial settings for starting CAN communications. The channels and the transfer rate used for the communication are specified in the arguments.

The following processes are required if the channel to be used has not been initialized.

- Initializing the variables used with the API functions
- Releasing the CAN module from the stop state
- Setting the ports to input or output
- Setting the CAN module to the global reset mode
- Setting the selected channel to the channel reset mode
- Initializing the CAN registers to be used for the CAN communication
- Specifying the transfer rate for the CAN communication

(5) **Returned values**

None

3.4.2 Stopping the CAN Module

R_CAN_Close

(1) **Synopsis**

Stops the CAN communication and releases the CAN module.

(2) **C language format**

void R_CAN_Close(uint32_t ch);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number

(4) **Description**

This function makes settings for ending the current CAN communication. When executed, the channels specified by the arguments are disabled.

The following operations are included:

- Setting the CAN module to the stop state
- Disabling the CAN interrupts

Call R_CAN_Open( ) (initialization function) to restart communication after this function has been called. If the ongoing communication is forcibly stopped, the communication is not guaranteed.

(5) **Returned values**

None

3.4.3 Making a Transition Between the Global Modes

R_CAN_GlobalControl

(1) **Synopsis**

Controls the RSCAN module as a whole.

(2) **C language format**

void R_CAN_GlobalControl(uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	mode	Global mode GL_MODE_OPE: Makes a transition to global operation mode. GL_MODE_RESET: Makes a transition to global reset mode. GL_MODE_STOP: Makes a transition to global stop mode. GL_MODE_TEST: Makes a transition to global test mode.

(4) **Description**

This function sets the global mode of the RSCAN module to whichever of the following global modes specified in the argument.

- Global stop mode: The clock for the whole module is stopped. Lower-power consumption is possible in this mode.
- Global reset mode: The initial settings for the whole RSCAN module are made in this mode.
- Global test mode: Tests (RAM test and inter-channels transfer test) are carried out in this mode.
- Global operation mode: Operation of the whole RSCAN module is enabled in this mode. The RSCAN module is normally in this mode.

(5) **Returned values**

None

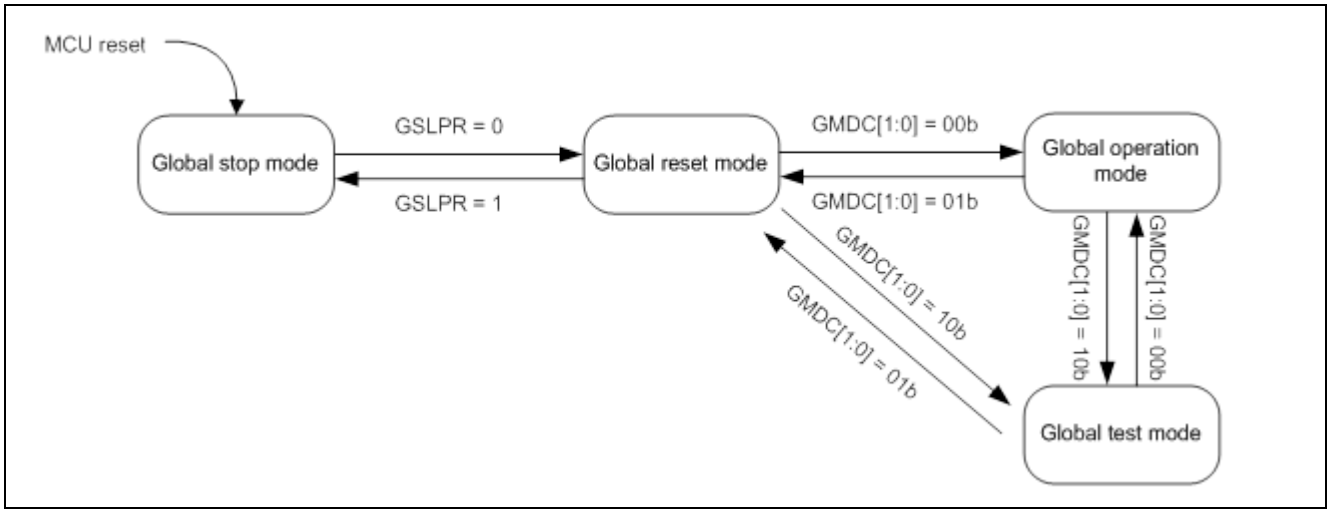


Figure 3-1 Transition Diagram Among Global Modes

3.4.4 Making a Transition Between the Channel Modes

R_CAN_ChannelControl

(1) **Synopsis**

Controls channels used for CAN communication.

(2) **C language format**

void R_CAN_ChannelControl(uint32_t ch, uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	uint32_t	mode	Channel mode CH_MODE_STOP: Makes a transition to channel stop mode. CH_MODE_RESET: Makes a transition to channel reset mode. CH_MODE_WAIT: Makes a transition to channel halt mode. CH_MODE_COMM: Makes a transition to channel transfer mode.

(4) **Description**

This function sets the state of the selected channel to whichever of the following channel modes specified in the argument:

- Channel stop mode: The clock for the specified channel is stopped in this mode.

- Channel reset mode: The initial settings for the channel is made in this mode.

- Channel halt mode: The CAN module is halted and tests for the specified channels are enabled.

- Channel transfer mode: CAN communications are handled in this mode. The CAN module is normally in this mode.

(5) **Returned values**

None

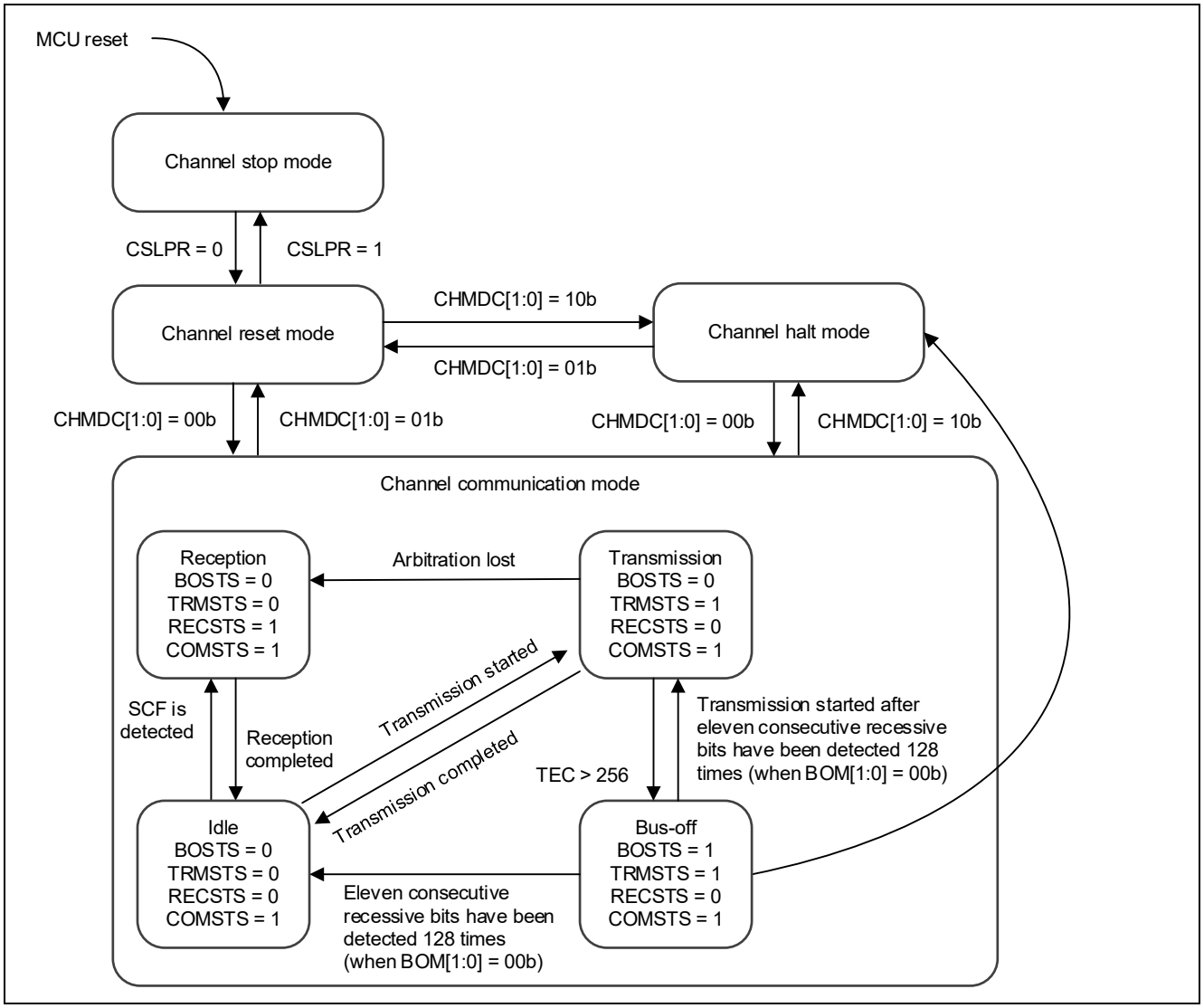


Figure 3-2 Transition Diagram among Channel Modes

3.4.5 Setting the Transfer Rate

R_CAN_SetBitrate

(1) **Synopsis**

Specifies the transfer rate for the CAN communication.

(2) **C language format**

void R_CAN_SetBitrate(uint32_t ch, uint32_t frequency);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	uint32_t	frequency	Transfer rate BAUD_RATE_1MBPS: 1 Mbps BAUD_RATE_500KBPS: 500 Kbps BAUD_RATE_125KBPS: 125 Kbps

(4) **Description**

This function specifies the transfer rate for the CAN communication by using the value set in the argument.

For details, refer to section 27.9.1.2, Bit Timing Setting and section 27.9.1.3, Communication Speed Setting, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

None

3.4.6 Registering the Information of the Buffers for Use in Transmission and Reception

R_CAN_UseBufferEntry

(1) **Synopsis**

Registers the information of the buffers for use in the CAN communications.

(2) **C language format**

void R_CAN_UseBufferEntry(can_used_buffer_t * obj);

(3) **Parameters**

I/O	Parameter	Description
I	can_used_buffer_t * obj	Pointer to the structure which holds the information related to buffers

(4) **Description**

This function registers the following information of the buffers for use in the CAN communications.

- Transmission buffer number
- Reception buffer number
- Reception FIFO buffer number
- Number of the transmission/reception FIFO buffer in transmission mode
- Number of the transmission/reception FIFO buffer in reception mode
- Number of the buffer to which the transmission/reception FIFO in transmission mode links.

(5) **Returned values**

None

3.4.7 Enabling the Reception FIFO Buffer

R_CAN_SetRxFifoBuffer

(1) **Synopsis**

Enables the reception FIFO buffers.

(2) **C language format**

void R_CAN_SetRxFifoBuffer(uint32_t ch, can_rfcc_t * obj);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	can_rfcc_t	* obj	Information of the reception FIFO buffer

(4) **Description**

This function enables reception of messages by using the reception FIFO buffers.

A CAN reception FIFO interrupt occurs on reception of a message. Register the reception FIFO buffer number to be used for the CAN communication by using the R_CAN_UseBufferEntry() function before calling this function.

The received messages are read by using the R_CAN_ReadRxFifo() function.

For details, refer to section 27.5, Reception Function and section 27.9.2.2, FIFO Buffer Reading Procedure, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

None

3.4.8 Enabling the Transmission/Reception FIFO Buffer

R_CAN_SetFifoBuffer

(1) **Synopsis**

Enables the transmission/reception FIFO buffers.

(2) **C language format**

void R_CAN_SetFifoBuffer(uint32_t ch, uint32_t mode, can_cfcc_t * obj);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	uint32_t	mode	Mode CAN_TX_MODE: Transmission mode CAN_RX_MODE: Reception mode
I	can_cfcc_t	* obj	Information of the transmission/reception FIFO buffer settings

(4) **Description**

This function enables transmission and reception of messages by using the transmission/reception FIFO buffers.

- Transmission mode

A transmission completion interrupt occurs on completion of transmission of a message from the transmission/reception FIFO buffer in transmission mode, with the source for the interrupt as completion of transmission.

For details, refer to section 27.6, Transmission Functions and section 27.9.3.2, Procedure for Transmission from Transmit/Receive FIFO Buffers, in the EC-1 User's Manual: Hardware.

- Reception mode

A reception completion interrupt occurs on completion of reception of a message at the transmission/reception FIFO buffer in reception mode. Register the transmission/reception FIFO buffers to be used for the CAN communication by using the R_CAN_UseBufferEntry() function before calling this function. The received messages are read by using the R_CAN_ReadFifo() function.

For details, refer to section 27.5, Reception Function and section 27.9.2.2, FIFO Buffer Reading Procedure, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

None

3.4.9 Releasing the Transmission/Reception FIFO Buffer

R_CAN_ReleaseFifoBuffer

(1) **Synopsis**

Releases the transmission/reception FIFO buffers used for the CAN communications.

(2) **C language format**

void R_CAN_ReleaseFifoBuffer(uint32_t ch, uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	uint32_t	mode	Mode CAN_TX_MODE: Transmission mode CAN_RX_MODE: Reception mode

(4) **Description**

This function releases the transmission/reception FIFO buffers used for the CAN communications.

(5) **Returned values**

None

3.4.10 Releasing the Reception FIFO Buffer

R_CAN_ReleaseRxFifoBuffer

(1) **Synopsis**

Releases the reception FIFO buffers used for the CAN communications.

(2) **C language format**

void R_CAN_ReleaseRxFifoBuffer(uint32_t ch);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number

(4) **Description**

This function releases the reception FIFO buffer used for the CAN communications.

(5) **Returned values**

None

3.4.11 Releasing the Transmission Buffer or the Reception Buffer

R_CAN_ReleaseBuffer

(1) **Synopsis**

Releases the buffers used for the CAN communications.

(2) **C language format**

void R_CAN_ReleaseBuffer(uint32_t ch, uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	uint32_t	mode	Mode CAN_TX_MODE: Transmission mode CAN_RX_MODE: Reception mode

(4) **Description**

This function releases the buffers used for the CAN communications.

(5) **Returned values**

None

3.4.12 Reading the Status of the Transmission Buffer

R_CAN_GetTxBufferStatus

(1) **Synopsis**

Reads the status of the transmission buffer.

(2) **C language format**

uint32_t R_CAN_GetTxBufferStatus(uint32_t ch);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number

(4) **Description**

This function is used for reading the status of the transmission buffer.

(5) **Returned values**

Returned Value	Meaning
0	No ongoing transmission
1	Transmission in progress

3.4.13 Writing Messages to Be Transmitted to the Transmission Buffer

R_CAN_WriteBuffer

(1) **Synopsis**

Writes messages to the transmission buffer.

(2) **C language format**

void R_CAN_WriteBuffer(uint32_t ch, can_tx_message_t * msg);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	can_tx_message_t	* msg	Information of messages for transmission

(4) **Description**

This function is used for writing messages to the transmission buffer.

The message ID, the data format, the data length, the label information, and the data to be transmitted are stored in the can_tx_message_t structure and passed to an argument of this function.

For details, refer to section 27.6, Transmission Functions and section 27.9.3.1, Procedure for Transmission from Transmit Buffers, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

None

3.4.14 Reading the Status of the Transmission/Reception FIFO Buffer

R_CAN_GetFifoStatus

(1) **Synopsis**

Reads the status of the transmission/reception FIFO buffer.

(2) **C language format**

uint32_t R_CAN_GetFifoStatus(uint32_t ch, uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	uint32_t	mode	Mode CAN_TX_MODE: Transmission mode CAN_RX_MODE: Reception mode

(4) **Description**

This function is used for reading the status of the transmission/reception FIFO buffer.

(5) **Returned values**

Returned Value	Meaning
0	Transmission/reception FIFO buffer is not full
1	Transmission/reception FIFO buffer is full

3.4.15 Writing Messages to Be Transmitted to the Transmission/Reception FIFO Buffer

R_CAN_WriteFifo

(1) **Synopsis**

Writes messages to the transmission/reception FIFO buffer in transmission mode.

(2) **C language format**

void R_CAN_WriteFifo(uint32_t ch, can_tx_message_t * msg);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	can_tx_message_t	* msg	Information of the message for transmission

(4) **Description**

This function is used for writing messages for transmission to the transmission/reception FIFO buffer in transmission mode.

The message ID, the data format, the data length, the label information, and the data to be transmitted are stored in the can_tx_message_t structure and passed to an argument of this function.

For details, refer to section 27.6, Transmission Functions and section 27.9.3.2, Procedure for Transmission from Transmit/Receive FIFO Buffers, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

None

3.4.16 Starting Transmission

R_CAN_Tx

(1) **Synopsis**

Starts transmission in CAN communication.

(2) **C language format**

void R_CAN_Tx(uint32_t ch, can_tx_message_t * msg);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	can_tx_message_t	* msg	Information of the message for transmission

(4) **Description**

This function starts transmission in CAN communication.

- Transmission by using the transmission buffer
Set the transmission request bit for the relevant transmission buffer to 1 (transmission is requested).

- Transmission by using the transmission/reception FIFO buffer in transmission mode

Set the transmission/reception FIFO buffer enable bit to 1 (transmission/reception FIFO buffers are used)

(5) **Returned values**

None

3.4.17 Making Settings for Reception

R_CAN_RxSet

(1) **Synopsis**

Enables reception.

(2) **C language format**

void R_CAN_RxSet(uint32_t ch, can_rx_rule_t * rule);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	can_rx_rule_t	* rule	Information of the reception rule

(4) **Description**

This function is used for setting rules for receiving messages.

The information of the reception rules are stored in the can_rx_rule_t structure and passed to an argument for the reception rule of this function.

For details, refer to section 27.5.1, Data Processing Using the Reception Rule Table and section 27.9.1.4, Reception Rule Setting, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

None

3.4.18 Reading Received Messages from the Reception Buffer

R_CAN_ReadBuff

(1) **Synopsis**

Reads received messages from the reception buffer.

(2) **C language format**

uint32_t R_CAN_ReadBuff(uint32_t ch, can_rx_message_t * obj);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	can_rx_message_t	* obj	Pointer to the area where the received messages are stored

(4) **Description**

This function is used for reading messages from the reception buffers.

For details, refer to section 27.5, Reception Function and section 27.9.2.1, Receive Buffer Reading Procedure, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

Returned Value	Meaning
CAN_OK	Data is successfully read from the reception buffer.
CAN_EMPTY	There are no new messages in the reception buffer.

3.4.19 Reading Received Messages from the Reception FIFO Buffer

R_CAN_ReadRxFifo

(1) **Synopsis**

Reads the received messages from the reception FIFO buffer.

(2) **C language format**

uint32_t R_CAN_ReadRxFifo(can_rx_message_t * obj);

(3) **Parameters**

I/O	Parameter	Description
I	can_rx_message_t * obj	Pointer to the area where the received messages are stored

(4) **Description**

This function is used for reading messages from the reception FIFO buffer.

For details, refer to section 27.5, Reception Function and section 27.9.2.2, FIFO Buffer Reading Procedure, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

Returned Value	Meaning
CAN_OK	Data are successfully read from the reception FIFO buffer.
CAN_EMPTY	There are no unread messages in the reception FIFO buffer (buffer empty).
CAM_LOST	FIFO message lost.

3.4.20 Reading Received Messages from the Transmission/Reception FIFO Buffer

R_CAN_ReadFifo

(1) **Synopsis**

Reads the received messages from the transmission/reception FIFO buffer.

(2) **C language format**

uint32_t R_CAN_ReadFifo(uint32_t ch, can_rx_message_t * obj);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	can_rx_message_t	* obj	Pointer to the area where the received messages are stored

(4) **Description**

This function is used for reading the messages from the transmission/reception FIFO buffer.

For details, refer to section 27.5, Reception Function and section 27.9.2.2, FIFO Buffer Reading Procedure, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

Returned Value	Meaning
CAN_OK	Data are successfully read from the transmission/reception FIFO buffer.
CAN_EMPTY	There are no unread messages in the transmission/reception FIFO buffer (buffer empty).
CAM_LOST	FIFO message lost.

3.4.21 Getting the Number of Unread Messages in the Transmission/Reception FIFO Buffer

R_CAN_GetFifoMessageNum

(1) **Synopsis**

Gets the number of unread messages from the transmission/reception FIFO buffer.

(2) **C language format**

uint32_t R_CAN_GetFifoMessageNum(uint32_t ch);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number

(4) **Description**

This function returns the number of unread messages in the transmission/reception FIFO buffer.

For details, refer to section 27.5, Reception Function and section 27.9.2.2, FIFO Buffer Reading Procedure, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

Returned Value	Meaning
	Number of unread messages

3.4.22 Getting the Number of Unread Messages in the Reception FIFO Buffer

R_CAN_GetRxFifoMessageNum

(1) **Synopsis**

Gets the number of unread messages in the transmission/reception FIFO buffer.

(2) **C language format**

uint32_t R_CAN_GetRxFifoMessageNum(void);

(3) **Parameter**

None

(4) **Description**

This function returns the number of unread messages in the transmission/reception FIFO buffer.

For details, refer to section 27.5, Reception Function and section 27.9.2.2, FIFO Buffer Reading Procedure, in the EC-1 User's Manual: Hardware.

(5) **Returned values**

Returned Value	Meaning
	Number of unread messages

3.4.23 Making Settings for Transfer Tests

R_CAN_SetCommTestMode

(1) **Synopsis**

Selects the transfer test mode.

(2) **C language format**

void R_CAN_SetCommTestMode(uint32_t ch, uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number
I	uint32_t	mode	Test mode CH_TEST_STANDARD: Standard test mode CH_TEST_LISTENONLY: Listen-only test mode CH_TEST_SELF0: Self-test mode 0 (external loopback mode) CH_TEST_SELF1: Self-test mode 1 (internal loopback mode)

(4) **Description**

This function is used for selecting a test mode.
The following tests modes are available:
- Standard test mode
- Listen-only test mode
- Self-test mode 0 (external loopback mode)
- Self-test mode 1 (internal loopback mode)

(5) **Returned values**

None

3.4.24 Releasing from the Test Mode and Entering the Channel Transfer Mode

R_CAN_ResetTestMode

(1) **Synopsis**

Clears the transfer test state.

(2) **C language format**

void R_CAN_ResetTestMode(uint32_t ch);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	ch	Channel number

(4) **Description**

This function clears the test mode set by using the R_CAN_SetCommTestMode() function.

After the test, always clear the state by calling this function.

(5) **Returned values**

None

3.4.25 Registering the Interrupt Handler

R_CAN_SetInterruptHandler

(1) **Synopsis**

Registers the callback function which is called by the interrupt handling routines used in the CAN communications.

(2) **C language format**

void R_CAN_SetInterruptHandler(uint32_t ch, can_callback_t * pcallback);

(3) **Parameters**

I/O	Parameters		Description
I	uint32_t	ch	Channel number
I	can_callback_t	* pcallback	Information of the callback function

(4) **Description**

This function is used to register the callback function which is called by an interrupt handling routine used in the CAN communications.

There are the following interrupt handling routines:

- CAN global error
- CAN0 error
- CAN1 error
- CAN reception FIFO
- CAN0 transmission/reception FIFO reception completion
- CAN0 transmission
- CAN1 transmission/reception FIFO reception completion
- CAN1 transmission

* pcallback

Pointer to the structure which holds the information of the callback function. Write the callback function name in the member of this structure. Write null if the pointer is not used.

(5) **Returned values**

None

3.4.26 Enabling or Disabling the CAN Module Interrupt Vectors

R_CAN_SetInterruptEnableDisable

(1) **Synopsis**

Enables or disables the interrupt handler used in CAN communications.

(2) **C language format**

void R_CAN_SetInterruptEnableDisable(uint32_t enable_disable);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	enable_disable	Enables or disables the interrupt handler used in CAN communications CAN_INTR_DISABLE: Disables the interrupt handler used in CAN communications CAN_INTR_ENABLE: Enables the interrupt handler used in CAN communications

(4) **Description**

This function enables or disables the interrupt handlers used in CAN communications.

There are the following interrupt handling routines:

- CAN global error
- CAN0 error
- CAN1 error
- CAN reception FIFO
- CAN0 transmission/reception FIFO reception completion
- CAN0 transmission
- CAN1 transmission/reception FIFO reception completion
- CAN1 transmission

(5) **Returned values**

None

3.4.27 Getting the Interrupt Source

R_CAN_GetInterruptSource

(1) **Synopsis**

Gets the interrupt source.

(2) **C language format**

void R_CAN_GetInterruptSource(can_intr_source_t * obj);

(3) **Parameters**

I/O	Parameter	Description
I	can_intr_source_t * obj	Information of the interrupt source

(4) **Description**

This function returns an indicator of the source of an interrupt.

- A state flag is set
- A port is set to input or output
- An I2C output port is allocated
- The RIIC module is released from the stop state
- The variables for use in this API function are initialized
- The RIIC registers for the communication with the RIIC module are initialized
- The RIIC interrupt is disabled

* obj

Pointer to the structure which holds the information of the interrupt source. Each interrupt source this function read is stored.

(5) **Returned values**

None

3.4.28 Clearing the Interrupt Source

R_CAN_ClearInterruptSource

(1) **Synopsis**

Clears the information of the interrupt source.

(2) **C language format**

void R_CAN_ClearInterruptSource(void);

(3) **Parameters**

None

(4) **Description**

This function clears the source for the corresponding interrupt.

(5) **Returned values**

None

3.5 CMT Control

This LSI provides drivers for controlling the compare match timer (CMT) function.

3.5.1 Initializing CMT Channels

R_CMT_Init

(1) **Synopsis**

Initializes CMT channels.

(2) **C language format**

int32_t R_CMT_Init(uint32_t channel, uint16_t cks);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify the CMT channel to be initialized. Settable range: 0 to 3
I	uint16_t	cks	Select the clock to be input to the CMCNTn counter. CMT_CKS_DIVISION_8: PCLKD / 8 CMT_CKS_DIVISION_32: PCLKD / 32 CMT_CKS_DIVISION_128: PCLKD / 128 CMT_CKS_DIVISION_512: PCLKD / 512

(4) **Description**

This function initializes the CMT channel specified in the argument.

The clock to be input to the CMCNTn counter is selected by the value specified in argument *cks*.

(5) **Returned values**

Returned Value	Meaning
CMT_SECCESS	Success
CMT_ERR	Error

3.5.2 Making Periodic Event Settings

R_CMT_CreatePeriodic

(1) **Synopsis**

Makes periodic event settings.

(2) **C language format**

int32_t R_CMT_CreatePeriodic(uint32_t channel, uint32_t frequency_hz, void (* callback)(uint32_t channnel));

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify the CMT channel for which periodic events are to be set. Settable range: 0 to 3
I	uint32_t	frequency_hz	Specify the intervals at which events are to be generated. (Unit: Hz)
I	void	(* callback)(uint32_t channnel)	Specify the pointer to the callback function executed when a CMT channel compare match interrupt is generated.

(4) **Description**

This function generates CMT channel compare match interrupts at the intervals specified by the argument.

It executes the callback function specified in argument *callback* when a CMT channel compare match interrupt is generated.

Count operation is continued even if a CMT channel compare match interrupt is generated.

Execute the periodic event stop function to stop the periodic events started by this function.

(5) **Returned values**

Returned Value	Meaning
CMT_SECCESS	Success
CMT_ERR	Error

Remarks Before executing this function, initialize the CMT channel by using the CMT channel initialization function.

3.5.3 Making One-Shot Event Settings

R_CMT_CreateOneShot

(1) **Synopsis**

Makes one-shot event settings.

(2) **C language format**

int32_t R_CMT_CreateOneShot(uint32_t channel, uint32_t period_us, void (* callback)(uint32_t channnel));

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify the CMT channel for which periodic events are to be set. Settable range: 0 to 3
I	uint32_t	period_us	Specify the intervals (in μs) at which events are to be generated.
I	void	(* callback)(uint32_t channnel)	Specify the pointer to the callback function executed when a CMT channel compare match interrupt is generated.

(4) **Description**

This function generates CMT channel compare match interrupts at the intervals specified by the argument.

It executes the callback function specified in argument *callback* when a CMT channel compare match interrupt is generated.

Count operation is continued even if a CMT channel compare match interrupt is generated.

Execute the periodic event stop function to stop the periodic events started by this function.

(5) **Returned values**

Returned Value	Meaning
CMT_SECCESS	Success
CMT_ERR	Error

Remarks Before executing this function, initialize the CMT channel by using the CMT channel initialization function.

3.5.4 Stopping CMT Operation

R_CMT_Stop

(1) **Synopsis**

Stops CMT operation.

(2) **C language format**

int32_t R_CMT_Stop(uint32_t channel);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify the CMT channel to be stopped. Settable range: 0 to 3

(4) **Description**

This function stops the CMT channel operation started by the periodic event setting or one-shot event setting.

(5) **Returned values**

Returned Value	Meaning
CMT_SECCESS	Success
CMT_ERR	Error

Remarks Before executing this function, initialize the CMT channel by using the CMT channel initialization function.

3.5.5 Initializing the CMT

userdef_cmt_init

(1) **Synopsis**

Initializes the CMT (user-defined).

(2) **C language format**

int32_t userdef_cmt_init (uint32_t channel, uint16_t cks);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify a CMT channel. Settable range: 0 to 3
I	uint16_t	cks	Select the clock to be input to the CMCNTn counter. CMT_CKS_DIVISION_8: PCLKD / 8 CMT_CKS_DIVISION_32: PCLKD / 32 CMT_CKS_DIVISION_128: PCLKD / 128 CMT_CKS_DIVISION_512: PCLKD / 512

(4) **Description**

This function initializes the CMT channel specified in the argument.

The clock to be input to the CMCNTn counter is selected by the value specified in argument *cks*.

(5) **Returned values**

Returned Value	Meaning
CMT_SECCESS	Success
CMT_ERR	Error

3.5.6 Making CMT Interval Settings

userdef_cmt_create

(1) **Synopsis**

Makes the CMT interval settings (user-defined).

(2) **C language format**

int32_t userdef_cmt_create (uint32_t channel, uint32_t frequency_hz, void (* callback)(uint32_t ch), uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify a CMT channel. Settable range: 0 to 3
I	uint32_t	frequency_hz	Specify the intervals at which events are to be generated. (Unit: Hz)
I	void	(* callback)(uint32_t ch)	Specify the pointer to the callback function executed when a CMT channel compare match interrupt is generated.
I	uint32_t	mode	Operating mode of the CMT channel CMT_MODE_PERIODIC: Periodic event setting CMT_MODE_ONESHOT: One-shot event setting

(4) **Description**

This function is used to make event interval and interrupt enable settings for each channel and register a callback function to be executed when an interrupt is generated.

(5) **Returned values**

Returned Value	Meaning
CMT_SECCESS	Success
CMT_ERR	Error

Remarks Before executing this function, initialize the CMT channel by using the CMT channel initialization function.

3.5.7 Stopping CMT Operation

userdef_cmt_stop

(1) **Synopsis**

Stops CMT operation (user-defined).

(2) **C language format**

int32_t userdef_cmt_stop (uint32_t channel);

(3) **Parameters**

I/O	Parameter	Description
I	uint32_t channel	Specify the CMT channel to be stopped. Settable range: 0 to 3

(4) **Description**

This function stops the CMT channel operation started by the periodic event setting or one-shot event setting.

(5) **Returned values**

Returned Value	Meaning
CMT_SECCESS	Success
CMT_ERR	Error

Remarks Before executing this function, initialize the CMT channel by using the CMT channel initialization function.

3.5.8 CMI Interrupt Handler

userdef_cmt_isr_cmi

(1) **Synopsis**

CMI interrupt handler (user-defined)

(2) **C language format**

int32_t userdef_cmt_isr_cmi (uint32_t channel);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify a CMT channel. Settable range: 0 to 3

(4) **Description**

This function executes the callback function for the specified CMT channel.

(5) **Returned values**

Returned Value	Meaning
CMT_SECCESS	Success
CMT_ERR	Error

Remarks Before executing this function, initialize the CMT channel by using the CMT channel initialization function.

3.5.9 CMI0 Interrupt Handler

cmt0_isr

(1) **Synopsis**

CMI0 interrupt handler

(2) **C language format**

void cmt0_isr (void);

(3) **Parameters**

None

(4) **Description**

This function executes the processing for an interrupt generated on channel 0 (interrupt source: compare match interrupt_ch0).

(5) **Returned values**

None

3.5.10 CMI1 Interrupt Handler

cmt1_isr

(1) **Synopsis**

CMI1 interrupt handler

(2) **C language format**

void cmt1_isr (void);

(3) **Parameters**

None

(4) **Description**

This function executes the processing for an interrupt generated on channel 1 (interrupt source: compare match interrupt_ch1).

(5) **Returned values**

None

3.5.11 CMI2 Interrupt Handler

cmt2_isr

(1) **Synopsis**

CMI2 interrupt handler

(2) **C language format**

void cmt2_isr (void);

(3) **Parameters**

None

(4) **Description**

This function executes the processing for an interrupt generated on channel 2 (interrupt source: compare match interrupt_ch0).

(5) **Returned values**

None

3.5.12 CMI3 Interrupt Handler

cmt3_isr

(1) **Synopsis**

CMI3 interrupt handler

(2) **C language format**

void cmt3_isr (void);

(3) **Parameters**

None

(4) **Description**

This function executes the processing for an interrupt generated on channel 3 (interrupt source: compare match interrupt_ch1).

(5) **Returned values**

None

3.6 ETHER Control

This LSI provides drivers for making communication settings for EtherCAT.

3.6.1 Initializing EtherCAT

EtherCAT_init

(1) **Synopsis**

Initializes EtherCAT.

(2) **C language format**

void EtherCAT_init(void);

(3) **Parameters**

None

(4) **Description**

This function performs initialization processing required for EtherCAT communication.

(5) **Returned values**

None

3.6.2 Initializing Ether Interrupt Requests

R_ETHER_IRQ_Init

(1) **Synopsis**

Initializes Ether interrupt requests.

(2) **C language format**

int32_t R_ETHER_IRQ_Init(void);

(3) **Parameters**

None

(4) **Description**

This function initializes Ether interrupt requests.

- EtherCAT Sync0 interrupt
- EtherCAT Sync1 interrupt
- EtherCAT interrupt

(5) **Returned values**

Returned Value	Meaning
ETHER_SECCESS	Success
ETHER_ERR	Error

3.6.3 Initializing Ether Interrupt Requests

ether_irq_init

(1) **Synopsis**

Initializes Ether interrupt requests.

(2) **C language format**

int32_t ether_irq_init(void);

(3) **Parameters**

None

(4) **Description**

This function initializes Ether interrupt requests.

- EtherCAT Sync0 interrupt
- EtherCAT Sync1 interrupt
- EtherCAT interrupt

(5) **Returned values**

Returned Value	Meaning
ETHER_SECCESS	Success
ETHER_ERR	Error

3.6.4 EtherCAT SYNC Signal Output Pin 0 Interrupt Handler

ecat_sync0_isr

(1) **Synopsis**

EtherCAT SYNC signal output pin 0 interrupt handler

(2) **C language format**

void ecat_sync0_isr(void);

(3) **Parameters**

None

(4) **Description**

This function performs interrupt processing (interrupt source: EtherCAT Sync0 interrupt).

(5) **Returned values**

None

3.6.5 EtherCAT SYNC Signal Output Pin 1 Interrupt Handler

ecat_sync1_isr

(1) **Synopsis**

EtherCAT SYNC signal output pin 1 interrupt handler

(2) **C language format**

void ecat_sync1_isr(void);

(3) **Parameters**

None

(4) **Description**

This function performs interrupt processing (interrupt source: EtherCAT Sync1 interrupt).

(5) **Returned values**

None

3.6.6 EtherCAT Interrupt Handler

ecat_isr

(1) **Synopsis**

EtherCAT interrupt handler

(2) **C language format**

void ecat_isr(void);

(3) **Parameters**

None

(4) **Description**

This function performs interrupt processing (interrupt source: EtherCAT interrupt).

(5) **Returned values**

None

3.7 RSPI Control

This LSI provides drivers for performing communication using the serial peripheral interface (RSPI).

3.7.1 Initializing RSPI Control

R_RSPIm_Create

(1) **Synopsis**

Initializes RSPI control.

(2) **C language format**

void R_RSPIm_Create(uint16_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	mode	Operating mode of RSPI channel m _RSPI_MSMODE_MASTER: Master mode _RSPI_MSMODE_SLAVE: Slave mode

(4) **Description**

This function performs initialization processing required for EtherCAT communication over channel m.

(5) **Returned values**

None

3.7.2 RSPI port setting initialization process

R_RSPIm_Pin_Init

(1) **Synopsis**

RSPI port setting initialization process.

(2) **C language format**

void R_RSPIm_Start (void);

(3) **Parameters**

None

(4) **Description**

This function starts RSPI communication over channel m.

(5) **Returned values**

None

3.7.3 Starting RSPI Communication

R_RSPIm_Start

(1) **Synopsis**

Start RSPI communication.

(2) **C language format**

void R_RSPIm_Start (void);

(3) **Parameters**

None

(4) **Description**

This function start RSPI communication over channel m.

(5) **Returned values**

None

3.7.4 Stopping RSPI Communication

R_RSPIm_Stop

(1) **Synopsis**

Stops RSPI communication.

(2) **C language format**

void R_RSPIm_Stop (void);

(3) **Parameters**

None

(4) **Description**

This function stops RSPI communication over channel m.

(5) **Returned values**

None

3.7.5 Getting the RSPI Communication Status

R_RSPI_GetState

(1) **Synopsis**

Gets the RSPI communication status.

(2) **C language format**

rspi_state_t R_RSPI_GetState (void);

(3) **Parameters**

None

(4) **Description**

This function returns the RSPI communication status.

(5) **Returned values**

Returned Value	Meaning
OVRF: Overrun error flag	0: Overrun error has not occurred 1: Overrun error has occurred
IDLNF: RSPI idle flag	0: RSPI is in idle state 1: RSPI is in transmission state
MODF: Mode fault error flag	0: Mode fault error has not occurred 1: Mode fault error has occurred
PERF: Parity error flag	0: Parity error has not occurred 1: Parity error has occurred
RX_END: Reception completion flag	0: Reception has not completed 1: Reception has completed

3.7.6 RSPI communication status initialization process

R_RSPIm_ClearState

(1) **Synopsis**

RSPI communication status initialization process.

(2) **C language format**

void R_RSPIm_ClearState (void);

(3) **Parameters**

None

(4) **Description**

Initializes the status of RSPI communication.

(5) **Returned values**

None

3.7.7 Starting RSPI Transmission and Reception

R_RSPIm_Send_Receive

(1) **Synopsis**

Starts RSPI transmission and reception.

(2) **C language format**

MD_STATUS R_RSPIm_Send_Receive (const uint32_t * tx_buf, uint16_t tx_num, uint32_t * rx_buf);

(3) **Parameters**

I/O	Parameter		Description
I	const uint32_t	* tx_buf	Pointer to the buffer which stores data for transmission
I	uint16_t	tx_num	Total number of data to be transmitted and received
O	uint32_t	* tx_buf	Pointer to the buffer which stores received data

(4) **Description**

This function starts RSPI transmission and reception over channel m.

(5) **Returned values**

Returned Value	Meaning
MD_OK	Normal termination
MD_ARGERROR	Specification of argument *tx_num* is invalid.

Remarks 1. This API function performs repeated RSPI transmission of 1-byte units from the buffer specified by argument *tx_buf* the number of times specified by argument *tx_num*.

Remarks 2. This API function performs repeated RSPI reception of 1-byte units to the buffer specified by argument *tx_buf* the number of times specified by argument *tx_num*.

Remarks 3. To perform RSPI transmission and reception, function R_RSPIm_Start must be called before this API function.

3.7.8 Transmission Completion Callback Function

r_rspim_callback_transmitend

(1) **Synopsis**

Processing when transmission ends.

(2) **C language format**

void r_rspim_callback_transmitend (void);

(3) **Parameters**

None

(4) **Description**

This function performs processing when transmission over channel m ends.

(5) **Returned values**

None

Remarks 1. In master mode, this API function is called as a callback routine of r_rspim_idle_interrupt.

Remarks 2. In slave mode, this API function is called as a callback routine when all data for transmission is stored in the transmission buffer by r_rspim_transmit_interrupt.

3.7.9 Reception Completion Callback Function

r_rspim_callback_receiveend

(1) **Synopsis**

Processing of a reception buffer full interrupt.

(2) **C language format**

void r_rspim_callback_receiveend (void);

(3) **Parameters**

None

(4) **Description**

This function performs processing of a reception buffer full interrupt on channel m.

(5) **Returned values**

None

Remarks This API function is called as a callback routine of r_rspim_receive_interrupt.

3.7.10 Error Callback Function

r_rspim_callback_error

(1) **Synopsis**

Processing of an RSPI error interrupt.

(2) **C language format**

void r_rspim_callback_error (uint8_t err_type);

(3) **Parameters**

I/O	Parameter		Description
O	uint8_t	err_type	Source of an RSPI error interrupt (x: Undefined) xxxx00x1B: Detection of an overrun error xxxx01x0B: Detection of a mode fault error xxxx10x0B: Detection of a parity error

(4) **Description**

This function performs processing of an RSPI error interrupt on channel m.

(5) **Returned values**

None

Remarks This API function is called as a callback routine of r_rspim_error_interrupt.

3.7.11 Transmission Buffer Empty Interrupt Handler

r_rspim_transmit_interrupt

(1) **Synopsis**

Processing of a transmission buffer empty interrupt.

(2) **C language format**

void r_rspim_transmit_interrupt (void);

(3) **Parameters**

None

(4) **Description**

This function performs processing of a transmission buffer empty interrupt on channel m.

(5) **Returned values**

None

Remarks This function is called as an interrupt handler for transmission buffer empty interrupts.

3.7.12 Reception Buffer Full Interrupt Handler

r_rspim_receive_interrupt

(1) **Synopsis**

Processing of a reception buffer full interrupt.

(2) **C language format**

void r_rspim_receive_interrupt (void);

(3) **Parameters**

None

(4) **Description**

This function performs processing of a reception buffer full interrupt on channel m.

(5) **Returned values**

None

Remarks This function is called as an interrupt handler for reception buffer full interrupts.

3.7.13 RSPI Error Interrupt Handler

r_rspim_error_interrupt

(1) **Synopsis**

Processing of an RSPI error interrupt

(2) **C language format**

void r_rspim_receive_interrupt (void);

(3) **Parameters**

None

(4) **Description**

This function performs processing of an RSPI error interrupt on channel m.

(5) **Returned values**

None

Remarks This function is called as an interrupt handler for RSPI error interrupts.

3.7.14 RSPI Idle Interrupt Handler

r_rspim_idle_interrupt

(1) **Synopsis**

Processing of an RSPI idle interrupt

(2) **C language format**

void r_rspim_idle_interrupt (void);

(3) **Parameters**

None

(4) **Description**

This function performs processing of an RSPI idle interrupt on channel m.

(5) **Returned values**

None

Remarks This function is called as an interrupt handler for RSPI idle interrupts.

3.8 SCIFA_UART Control

This LSI provides drivers for controlling the UART function of the serial communication interface with FIFO (SCIFA).

3.8.1 Registering the SCIFA Interrupt Handler

R_SCIFA_SetInterruptHandler

(1) **Synopsis**

Registers the SCIFA interrupt handler.

(2) **C language format**

void R_SCIFA_SetInterruptHandler(scifa_callback_t * pcallback);

(3) **Parameters**

I/O	Parameter		Description
I	scifa_callback_t	* pcallback	Callback function

(4) **Description**

This function registers the SCIFA interrupt handler (interrupt source: Reception FIFO data full (RFD)).

(5) **Returned values**

None

3.8.2 Enabling SCIFA Reception Interrupts

R_SCIFA_EnableItrReg

(1) **Synopsis**

Enables SCIFA reception interrupts.

(2) **C language format**

void R_SCIFA_EnableItrReg(void);

(3) **Parameters**

None

(4) **Description**

This function enables the interrupt handler when the SCIFA is receiving data.

(5) **Returned values**

None

3.8.3 Disabling SCIFA Reception Interrupts

R_SCIFA_ResetItrReg

(1) **Synopsis**

Disables SCIFA reception interrupts.

(2) **C language format**

void R_SCIFA_ResetItrReg(void);

(3) **Parameters**

None

(4) **Description**

This function disables the interrupt handler when the SCIFA is receiving data.

(5) **Returned values**

None

3.8.4 Initializing SCIFA Channel UART Mode

R_SCIFA_UART_Init

(1) **Synopsis**

Initializes the SCIFA.

(2) **C language format**

int32_t R_SCIFA_UART_Init(uint32_t channel, uint32_t mode, uint16_t cks, uint8_t brr);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify the SCIFA channel to be initialized. Settable range: 0 to 4
I	uint32_t	mode	Specify the operating mode of the SCIFA. SCIFA_UART_MODE_R: Reception mode SCIFA_UART_MODE_W: Transmission mode SCIFA_UART_MODE_RW: Transmission/reception mode
I	uint16_t	cks	Select the clock source of the baud rate generator of the SCIFA. SCIFA_UART_CKS_DIVISION_1: SERICLK SCIFA_UART_CKS_DIVISION_4: SERICLK / 4 SCIFA_UART_CKS_DIVISION_16: SERICLK / 16 SCIFA_UART_CKS_DIVISION_64: SERICLK / 64
I	uint8_t	brr	Specify the value to be set in the SCIFA bit rate register (BRR). Settable range: Refer to the Hardware Manual.

(4) **Description**

This function initializes the SCIFA to asynchronous communication mode and sets ports used for the SCIFA.

(5) **Returned values**

Returned Value	Meaning
SCIFA_UART_SUCCESS	Initialization succeeded
SCIFA_UART_ERR	Argument error

Remarks 1. Only SCIFA channel 0 is installed. Channels 1 to 4 are not installed.

Remarks 2. SCIFA channel 0 uses the following pins:

TXD0: P23

RXD0: P42

3.8.6 Starting SCIFA Channel UART Mode

R_SCIFA_UART_Open

(1) **Synopsis**

Starts SCIFA channel UART mode.

(2) **C language format**

int32_t R_SCIFA_UART_Open(uint32_t channel, uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify an SCIFA channel. Settable range: 0 to 4
I	uint32_t	mode	Specify the operating mode of the SCIFA. SCIFA_UART_MODE_R: Reception mode SCIFA_UART_MODE_W: Transmission mode SCIFA_UART_MODE_RW: Transmission/reception mode

(4) **Description**

This function starts up the SCIFA in the operation mode (transmit, receiver or transmission/reception) specified by an argument and starts asynchronous communication.

(5) **Returned values**

Returned Value	Meaning
SCIFA_UART_SUCCESS	Started the SCIFA successfully
SCIFA_UART_ERR	Argument error

Remarks Only SCIFA channel 0 is installed. Channels 1 to 4 are not installed.

3.8.6 Receiving Data in SCIFA Channel UART Mode

R_SCIFA_UART_Receive

(1) **Synopsis**

Data reception processing

(2) **C language format**

int32_t R_SCIFA_UART_Receive(uint32_t channel, uint8_t *data);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify an SCIFA channel. Settable range: 0 to 4
I	uint8_t	*data	Specify the area to store the received data.

(4) **Description**

This function receives 1-byte data in RS-232C interface COM port communication.

(5) **Returned values**

Returned Value	Meaning
SCIFA_UART_SUCCESS	Reception succeeded
SCIFA_UART_ERR	Argument error
SCIFA_UART_ERR_RECEIVE	Reception error

Remarks Only SCIFA channel 0 is installed. Channels 1 to 4 are not installed.

3.8.7 Transmitting Data in SCIFA Channel UART Mode

R_SCIFA_UART_Send

(1) **Synopsis**

Data transmission processing

(2) **C language format**

int32_t R_SCIFA_UART_Send(uint32_t channel, uint8_t data);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	channel	Specify an SCIFA channel. Settable range: 0 to 4
I	uint8_t	data	Specify data for transmission.

(4) **Description**

This function transmits 1-byte data in RS-232C interface COM port communication.

(5) **Returned values**

Returned Value	Meaning
SCIFA_UART_SUCCESS	Transmission succeeded
SCIFA_UART_ERR	Argument error

Remarks Only SCIFA channel 0 is installed. Channels 1 to 4 are not installed.

3.8.8 Initializing SCIFA Channel 0 UART Mode

userdef_scifa0_uart_init

(1) **Synopsis**

Initializes SCIFA channel 0 UART mode (user-defined).

(2) **C language format**

void userdef_scifa0_uart_init(uint32_t mode, uint16_t cks, uint8_t brr);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	mode	Specify the operating mode of the SCIFA. SCIFA_UART_MODE_R: Reception mode SCIFA_UART_MODE_W: Transmission mode SCIFA_UART_MODE_RW: Transmission/reception mode
I	uint16_t	cks	Select the clock source of the baud rate generator of the SCIFA. SCIFA_UART_CKS_DIVISION_1: SERICLK SCIFA_UART_CKS_DIVISION_4: SERICLK / 4 SCIFA_UART_CKS_DIVISION_16: SERICLK / 16 SCIFA_UART_CKS_DIVISION_64: SERICLK / 64
I	uint8_t	brr	Specify the value to be set in the SCIFA bit rate register (BRR). Settable range: Refer to the Hardware Manual.

(4) **Description**

This function initializes the UART function of SCIFA channel 0.

(5) **Returned values**

None

3.8.9 Starting SCIFA Channel 0 UART Mode

userdef_scifa0_uart_open

(1) **Synopsis**

Starts SCIFA channel 0 UART mode (user-defined).

(2) **C language format**

void userdef_scifa0_uart_open(uint32_t mode);

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	mode	Specify the operating mode of the SCIFA. SCIFA_UART_MODE_R: Reception mode SCIFA_UART_MODE_W: Transmission mode SCIFA_UART_MODE_RW: Transmission/reception mode

(4) **Description**

This function opens the UART function of SCIFA channel 0.

Specify transmission, reception, or transmission/reception mode with argument *mode*.

(5) **Returned values**

None

3.8.10 Receiving Data in SCIFA Channel 0 UART Mode

userdef_scifa0_uart_receive

(1) **Synopsis**

Receives data over SCIFA channel 0 (user-defined).

(2) **C language format**

int32_t userdef_scifa0_uart_receive(uint8_t *data);

(3) **Parameters**

I/O	Parameter	Description
I	uint8_t *data	Specify the area to store the received data.

(4) **Description**

This function reads data from the UART function on SCIFA channel 0.

Upon a reception error, it clears the reception error state and set a reception error as the returned value.

When getting received data, it stores the received data in the area specified in the argument.

(5) **Returned values**

Returned Value	Meaning
SCIFA_UART_SUCCESS	Reception succeeded
SCIFA_UART_ERR_RECEIVE	Reception error

3.8.11 Transmitting Data in SCIFA Channel 0 UART Mode

userdef_scifa0_uart_send

(1) **Synopsis**

Transmits data over SCIFA channel 0 (user-defined).

(2) **C language format**

int32_t userdef_scifa0_uart_send(uint8_t *data);

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	data	Specify data for transmission.

(4) **Description**

This function writes data to the UART function on SCIFA channel 0.
It transmits the data specified in the argument.

(5) **Returned values**

None

3.8.12 Outputting Character Strings

__write

(1) **Synopsis**

Outputs character strings.

(2) **C language format**

size_t __write(int handle, const unsigned char * buffer, size_t size);

(3) **Parameters**

I/O	Parameter		Description
I	int	handle	File number to be read STDIN (0) STDOUT (1) STDERR (2)
O	const unsigned char	* buffer	Pointer to the area where read data is stored
I	size_t	size	Number of bytes to be read

(4) **Description**

By using serial communication by UART, this function transmits the data that is read in 1-byte units from the buffer specified in the argument.

(5) **Returned values**

Returned Value	Meaning
>=0	Number of characters sent
-1	File number error

3.8.13 Inputting Character Strings

__read

(1) **Synopsis**

Inputs character strings.

(2) **C language format**

size_t __read(int handle, unsigned char * buffer, size_t size);

(3) **Parameters**

I/O	Parameter		Description
I	int	handle	File number to be read STDIN (0) STDOUT (1) STDERR (2)
O	unsigned char	* buffer	Pointer to the area where read data is stored
I	size_t_t	size	Number of bytes read

(4) **Description**

By using serial communication by UART, this function receives the data by the buffer specified in the argument in 1-byte units and outputs the read data.

(5) **Returned values**

Returned Value	Meaning
>0	Number of received characters
-1	File number or received data error

3.8.14 Initializing Input/Output (Initializing SCIFA Channel 0)

IoInitScifa0

(1) **Synopsis**

Initializes input/output (initializes SCIFA channel 0).

(2) **C language format**

void IoInitScifa0 (void);

(3) **Parameters**

None

(4) **Description**

This function initializes SCIFA channel 0 to use the UART as standard input/output.

(5) **Returned values**

None

3.8.15 Getting Characters

loGetchar

(1) **Synopsis**

Gets characters.

(2) **C language format**

int32_t loGetchar (void);

(3) **Parameters**

None

(4) **Description**

This function receives a character from SCIFA channel 0 and returns data.
The character is loaded as the unsigned char type and converted to the int type.
This API function waits until it receives data.

(5) **Returned values**

Returned Value	Meaning
>0	Number of received bytes
-1	Error

3.8.16 Outputting Characters

ioPutchar

(1) **Synopsis**

Outputs characters.

(2) **C language format**

void ioPutchar (int32_t buffer)

(3) **Parameters**

I/O	Parameter		Description
I	int32_t	buffer	Output character

(4) **Description**

This function converts the character specified in the argument to the unsigned char type and outputs it to SCIFA channel 0.

It waits until it is ready to transmit.

(5) **Returned values**

None

3.9 Serial Flash ROM Control

This LSI provides drivers for controlling serial flash ROM.

3.9.1 Initializing Serial Flash ROM Control

R_SFLASH_init

(1) **Synopsis**

Initializes serial flash ROM control.

(2) **C language format**

int32_t R_SFLASH_init(void)

(3) **Parameters**

None

(4) **Description**

This function initializes serial flash ROM control.

(5) **Returned values**

Returned Value	Meaning
SFLASH_ER_OK	Success
SFLASH_ER_PARAM	Parameter error

3.9.2 Writing Data to Serial Flash ROM

R_SFLASH_program

(1) **Synopsis**

Writes data to serial flash ROM.

(2) **C language format**

int32_t R_SFLASH_program(uint8_t* buf, uint32_t addr, uint32_t size);

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	* buf	Pointer to the buffer for storing the write data
I	uint32_t	addr	Write start address
I	uint32_t	size	Write data size

(4) **Description**

This function writes data in the buffer to the area specified in argument *addr* in serial flash ROM by the size specified in argument *size*.

(5) **Returned values**

Returned Value	Meaning
SFLASH_ER_OK	Success
SFLASH_ER_PARAM	Parameter error

3.9.3 Erasing Data from Serial Flash ROM

R_SFLASH_erase

(1) **Synopsis**

Erases data from serial flash ROM.

(2) **C language format**

int32_t R_SFLASH_erase(uint32_t addr, uint32_t size)

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	addr	Data erase start address
I	uint32_t	size	Data erase size

(4) **Description**

This function erases data from the area specified in argument *addr* in serial flash ROM in the erasable unit including the area for the size specified in argument *size*.

(5) **Returned values**

Returned Value	Meaning
SFLASH_ER_OK	Success
SFLASH_ER_PARAM	Parameter error

3.10 USB Function Control

This LSI provides drivers for controlling the USB function.

3.10.1 Initializing the USB Module

R_USB_Open

(1) **Synopsis**

Initializes the USB module.

(2) **C language format**

void R_USB_Open(void)

(3) **Parameters**

None

(4) **Description**

This function initializes the USB module and its pin settings.
The USB function becomes available by calling this API function.

(5) **Returned values**

None

Remarks 1. Call this API function after calling R_usb_pstd_DriveRegistration().
Remarks 2. Let the user application call this API function only once during initialization.

Example of Use

```
void usb_smp_task()
{
        USB_PCDREG_t driver;
        :
        R_usb_pstd_DriverRegistration(&driver);
        R_USB_Open();          /* Start USB module */
        :
}
```

3.10.2 Requesting Data Transfer

R_usb_pstd_TransferStart

(1) **Synopsis**

Requests data transfer.

(2) **C language format**

USB_ER_t R_usb_pstd_TransferStart(USB_UTR_t *ptr)

(3) **Parameters**

I/O	Parameter	Description
I	USB_UTR_T *ptr	USB communication structure

(4) **Description**

This API function requests the peripheral control driver (PCD) of data transfer. Upon this request, the PCD carries out data transfer based on the transfer information set in argument *ptr* (USB_UTR_t structure).

Set the following information in the USB_UTR_t structure.

uint16_t keyword: Pipe number

USB_UTR_CB_t complete: Callback function

void* tranadr: Start pointer of the data storing buffer

uint32_t tranlen: Data transfer size

uint16_t status: Status

uint16_t pipectr: Not used

The callback function is called when data transfer is finished (specified data size or a short packet reception error occurred).

The remaining data length, status and transfer end information of the transmission or reception are set in argument *ptr* in the callback function.

For details about the callback function, see section 7.8.5, PCD Callback Functions.

(5) **Returned values**

Returned Value	Meaning
USB_OK	Success
USB_ERROR	Error
USB_QOVR	The specified pipe is processing data transfer.

Remarks 1. Make the user application or class driver call this API function.

Remarks 2. If the received data is n times the max packet size and it is less than the expected size, it is determined that data transfer is still in progress, so no callback occurs.

Example of Use

```
USB_UTR_t usb_smpl_trn_utr[USB_MAXPIPE_NUM + 1];

USB_ER_t usb_smp_task(uint16_t pipe, uint32_t size,uint8_t *table, USB_UTR_CB_t complete)
{
        /* Setting the transfer information */
        usb_smpl_trn_utr[pipe].keyword = pipe;                  /* Pipe number */
        usb_smpl_trn_utr[pipe].tranadr = table;                 /* Start pointer of the data storing buffer */
        usb_smpl_trn_utr[pipe].tranlen = size;          /* Data transfer size */
        usb_smpl_trn_utr[pipe].complete = complete;             /* Callback function */
        usb_smpl_trn_utr[pipe].segment = USB_TRAN_END;  /* Status */

        /* Requesting transfer start */
        err = R_usb_pstd_TransferStart(&usb_smpl_trn_utr[pipe]);

        return err;
}
```

3.10.3 Requesting Data Transfer Forced End

R_usb_pstd_TransferEnd

(1) **Synopsis**

Requests data transfer forced end.

(2) **C language format**

USB_ER_t R_usb_pstd_TransferEnd(uint16_t pipe)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	pipe	Pipe number

(4) **Description**

This API function requests the peripheral control driver (PCD) of forced end of data transfer. Upon this request, the PCD ends data transfer forcibly.

Upon forced end of data transfer, the callback function set when data transfer is requested (R_usb_pstd_TransferStart) is called.

The remaining data length, status and forced transfer end information of the transmission or reception are set in argument *ptr* in the callback function.

(5) **Returned values**

Returned Value	Meaning
USB_OK	Success
USB_ERROR	Error

Remarks 1. Make the user application or class driver call this API function.

Example of Use

```
void usb_smp_task()
{
                :
    /* Requesting transfer end */
    err = R_usb_pstd_TransferEnd(pipe);
                :
}
```

3.10.4 Making a Transition Between USB Device States

R_usb_pstd_ChangeDeviceState

(1) **Synopsis**

Makes a transition between USB device states.

(2) **C language format**

void R_usb_pstd_ChangeDeviceState(uint16_t state, uint16_t keyword, USB_UTR_CB_t complete)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	state	Device state to be transitioned to
I	uint16_t	keyword	Pipe number
I	USB_UTR_CB_t	complete	Callback function

(4) **Description**

A change of the USB device state can be requested to the peripheral control driver (PCD) by setting the following values in argument *state* and calling this API function.

・USB_DO_REMOTEWAKEUP

Makes a remote wakeup execution request to the PCD.

・USB_DP_ENABLE

Makes a D+ line pull-up request to the PCD.

・USB_DP_DISABLE

Makes a D+ line pull-up release request to the PCD.

・USB_DO_STALL

Makes a STALL response execution request to the PCD.

(5) **Returned values**

None

Remarks 1. Make the user application or class driver call this API function.

Remarks 2. If detecting a connection or disconnection interrupt, the firmware pulls up the D+ line or releases the pull-up automatically.

Remarks 3. The pipe number and the callback function are used only for a STALL response (USB_DO_STALL).

Example of Use

```
void usb_smp_task()
{
                :
        /* Requesting a STALL response execution */
        R_usb_pstd_ChangeDeviceState(USB_DO_STALL, USB_PIPE1, &usb_smp_dummy)
                :
}
```

3.10.5 Registering the Peripheral Device Class Driver (PDCD)

R_usb_pstd_DriverRegistration

(1) **Synopsis**

Registers the peripheral device class driver (PDCD).

(2) **C language format**

void R_usb_pstd_DriverRegistration(USB_PCDREG_t *registinfo)

(3) **Parameters**

I/O	Parameter	Description
I	USB_PCDREG_t *registinfo	Class driver structure

(4) **Description**

This function registers in the peripheral control driver (PCD) the information about the peripheral device class driver (PDCD) registered in the class driver structure.

(5) **Returned values**

None

Remarks 1. Let the user application call this API function only once during initialization.

Remarks 2. Only one device can be registered. For registered information, see Table 3-22 USB_PCDREG_t Structure.

Example of Use

```
void usb_smp_registration()
{
        USB_PCDREG_t driver;
                :
        /* Pipe Define Table address */
        driver.pipetbl = (uint16_t**)&usb_gpvendor_smpl_epptr;
        /* Device descriptor Table address */
        driver.devicetbl = (uint8_t*)&usb_gpvendor_smpl_DeviceDescriptor;
        /* Qualifier descriptor Table address */
        driver.qualitbl = (uint8_t*)&usb_gpvendor_smpl_QualifierDescriptor;
        /* Configuration descriptor Table address */
        driver.configtbl = (uint8_t**)&usb_gpvendor_smpl_ConPtr;
        /* Other configuration descriptor Table address */
        driver.othertbl = (uint8_t**)&usb_gpvendor_smpl_ConPtrOther;
        /* String descriptor Table address */
        driver.stringtbl = (uint8_t**)&usb_gpvendor_str_ptr;
        /* Device default */
        driver.devdefault = &usb_smpl_devdefault;
        /* Device configured */
        driver.devconfig = &usb_smpl_devconfig;
        /* Device detach */
        driver.devdetach = &usb_smpl_detach;
        /* Device suspend */
        driver.devsuspend = &usb_smpl_dummy_cb;
        /* Device resume */
        driver.devresume = &usb_smpl_dummy_cb;
        /* Interfaced change */
        driver.interface = &usb_smpl_dummy_cb;
        /* Control Transfer */
        driver.ctrltrans = (USB_CB_TRN_t)&usb_smpl_dummy_cb;
        /* Driver registration */
        R_usb_pstd_DriverRegistration(&driver);
}
```

3.10.6 Setting the PID Bits for the Specified Pipe to BUF

R_usb_cstd_SetBuf

(1) **Synopsis**

Sets the PID bits for the specified pipe to BUF.

(2) **C language format**

void R_usb_cstd_SetBuf(uint16_t pipe)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	pipe	Pipe number (only USB_PIPE0 can be set)

(4) **Description**

This function sets the PID bits for the specified pipe to BUF.

(5) **Returned values**

None

Remarks 1. Call this function in the setup stage processing of control transfer.

Example of Use

```
void usb_smp_task()
{
                :
        R_usb_cstd_SetBuf(USB_PIPE0);
                :
}
```

3.10.7 Setting the PID Bits for the Specified Pipe to STALL

R_usb_pstd_SetPipeStall

(1) **Synopsis**

Sets the PID bits for the specified pipe to STALL.

(2) **C language format**

void R_usb_pstd_SetPipeStall(uint16_t pipe)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	pipe	Pipe number (only USB_PIPE0 can be set)

(4) **Description**

This function sets the PID bits for the pipe specified in the argument to STALL.

(5) **Returned values**

None

Remarks 1. Call this function in the setup stage or status stage processing of control transfer.

Example of Use

```
void usb_smp_task()
{
                :
        R_usb_pstd_SetPipeStall(pipe);
                :
}
```

3.10.8 Requesting Data Transfer for Control IN Transfer

R_usb_pstd_ControlRead

(1) **Synopsis**

Requests data transfer for control IN transfer.

(2) **C language format**

uint16_t R_usb_pstd_ControlRead(uint32_t bsize, uint8_t *table)

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	bsize	Size of the buffer for storing data for transmission
I	uint8_t	*table	Address of the buffer for storing data for transmission

(4) **Description**

By calling this API function in the data stage of control IN transfer, the peripheral control driver (PCD) is requested to carry out data transfer.

After reading data from the area specified by argument **table*, the PCD writes it to the FIFO buffer and notifies the following result via the returned value (end_flag).

・USB_WRITESHRT

Data write finished (short packet data write)

・USB_WRITEEND

Data write finished (no continued data or packet transmission of data length 0)

・USB_WRITING

Data being written (there is continued data)

・USB_FIFOERROR

An FIFO access error occurred.

(5) **Returned values**

Returned Value	Meaning
end_flag	Data transfer result

Remarks 1. Call this API function in the data stage of control IN transfer. For details, see section 7.9.2, Peripheral Control Transfer.

Example of Use

```
uint8_t usb_smp_buf;

void usb_smp_task()
{
                    :
        /* Read 1-byte data to usb_smp_buf */
        R_usb_pstd_ControlRead((uint32_t)1, (uint8_t*)(&usb_smp_buf));
                    :
}
```

3.10.9 Requesting Data Transfer for Control OUT Transfer

R_usb_pstd_ControlWrite

(1) **Synopsis**

Requests data transfer for control OUT transfer.

(2) **C language format**

void R_usb_pstd_ControlWrite(uint32_t bsize, uint8_t *table)

(3) **Parameters**

I/O	Parameter		Description
I	uint32_t	bsize	Size of the data to be written to the buffer
I	uint8_t	*table	Address of the buffer to which data is written

(4) **Description**

By calling this API function in the data stage of control OUT transfer, the peripheral control driver (PCD) is requested to carry out data transfer.

The PCD reads data from the FIFO buffer and writes it to the area specified by argument **table*.

(5) **Returned values**

None

Remarks 1. Call this API function in the data stage of control OUT transfer. For details, see section 7.9.2, Peripheral Control Transfer.

Example of Use

```
uint8_t usb_smp_buf;

void usb_smp_task()
{
                :
    /* Read 1-byte data to usb_smp_buf */
    R_usb_pstd_ControlWrite((uint32_t)1, (uint8_t*)(&usb_smp_buf));
                :
}
```

3.10.10 Requesting Control Transfer End

R_usb_pstd_ControlEnd

(1) **Synopsis**

Requests control transfer end.

(2) **C language format**

void R_usb_pstd_ControlEnd(uint16_t status)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	status	Status

(4) **Description**

By calling this API function in the status stage of control transfer, the peripheral control driver (PCD) is requested to execute the status stage.

Set any of the following values in argument *status*.

- USB_CTRL_END

Ends the status stage normally.

- USB_DATA_ERR

Returns STALL to the host in the status stage.

(5) **Returned values**

None

Remarks 1. Call this API function in the data stage of control IN transfer. For details, see section 7.9.2, Peripheral Control Transfer.

Example of Use

```
uint8_t usb_smp_buf;

void usb_smp_task()
{
                :
    /* Read 1-byte data to usb_smp_buf */
    R_usb_pstd_ControlEnd(USB_CTRL_END);
                :
}
```

3.10.11 USB Interrupt Processing

R_usb_pstd_poll

(1) **Synopsis**

USB interrupt processing

(2) **C language format**

void R_usb_pstd_poll(void)

(3) **Parameters**

None

(4) **Description**

This function determines if a USB interrupt has occurred. It controls the hardware if a USB interrupt has occurred.

(5) **Returned values**

None

Remarks 1. Call this API function periodically after calling R_USB_Open().

Example of Use

```
uint8_t usb_smp_buf;

void usb_smp_task()
{
                :
        while(1)
        {
                R_usb_pstd_poll();
        }
                :
}
```

3.10.12 USB Transmission Processing

R_usb_pcdc_SendData

(1) **Synopsis**

USB transmission processing

(2) **C language format**

void R_usb_pcdc_SendData(uint8_t *table, uint32_t size, USB_UTR_CB_t complete)

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	table	Transfer data address
I	uint32_t	size	Transfer size
I	USB_UTR_CB_t	complete	Callback function for notifying processing end

(4) **Description**

This function sends data from the specified address by the specified transfer size.

After data has been sent, callback function complete is called.

(5) **Returned values**

None

Remarks 1. The result of USB transmission processing can be obtained from the argument of the callback function. For details, see Table 3-21 USB_UTR_t Structure and Table 3-22 USB_PCDREG_t Structure.

Example of Use

```
void usb_apl_task( void )
{
        uint8_t send_data[] = {0x01,0x02,0x03,0x04,0x05};       /* USB data for transmission*/
        uint32_t size = 5;                                      /* Number of USB data for transmission*/

        R_usb_pcdc_SendData(send_data, size, &usb_complete);
}

/* Callback function for notifying USB transmission completion */
void usb_complete(USB_UTR_t *mess)
{
        /* Describe the processing to be performed when USB transmission has been completed */
}
```

3.10.13 USB Reception Processing

R_usb_pcdc_ ReceiveData

(1) **Synopsis**

USB reception processing

(2) **C language format**

void R_usb_pcdc_ReceiveData (uint8_t *table, uint32_t size, USB_UTR_CB_t complete)

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	table	Transfer data address
I	uint32_t	size	Transfer size
I	USB_UTR_CB_t	complete	Callback function for notifying processing end

(4) **Description**

This function makes a USB reception request to the HCD.

When data of the specified transfer size has been received or data less than the max packet size has been received, callback function complete is called.

The USB received data is stored in the area specified by the transfer data address.

(5) **Returned values**

None

Remarks 1. The result of USB transmission processing can be obtained from the argument of the callback function. For details, see Table 3-21 USB_UTR_t Structure and Table 3-22 USB_PCDREG_t Structure.

Example of Use

```
void usb_apl_task( void )
{
        uint8_t receive_data[64];                   /* USB received data storage area */
        uint32_t size = 64;                         /* USB reception requested size */

        R_usb_pcdc_ReceiveData(receive_data, size, &usb_complete);
}

/* Callback function for notifying USB reception completion */
void usb_complete(USB_UTR_t *mess)
{
        /* Describe the processing to be performed when USB reception has been completed */
}
```

3.10.14 Sending class notification "SerialState"

R_usb_pcdc_SerialStateNotification

(1) **Synopsis**

Sends class notification “SerialState”

(2) **C language format**

void R_usb_pcdc_SerialStateNotification(USB_SCI_SerialState_t serial_state, USB_UTR_CB_t complete)

(3) **Parameters**

I/O	Parameter		Description
I	USB_SCI_SerialState_t	serial_state	Serial status
I	USB_UTR_CB_t	complete	Callback function for notifying processing end

(4) **Description**

This function sends CDC class notification “SerialState” to the USB host.

This transmission uses Interrupt IN Transfer.

After transmission has been completed, callback function complete is called.

(5) **Returned values**

None

Remarks 1. For details about USB_SCI_SerialState_t, see Table 3-23 USB_SCI_SerialState_t Structure.

Remarks 2. The result of USB transmission processing can be obtained from the argument of the callback function. For details, see Table 3-21 USB_UTR_t Structure and Table 3-22 USB_PCDREG_t Structure.

Example of Use

```
void usb_apl_task( void )
{
        USB_SCI_serialState_t serial_state; /* Serial status */

        serial_state.WORD = 0x0000;
        serial_state.bParity = 0x0020; /* D5:parity error */
        R_usb_pcdc_SerialStateNotification(serial_state, &usb_complete);
}

/* Callback function for notifying serial state transmission completion */
void usb_complete(USB_UTR_t *mess)
{
        /* Describe the processing to be performed when serial state transmission has been completed */
}
```

3.10.15 Control Transfer Processing for CDC

R_usb_pcdc_ctrltrans

(1) **Synopsis**

Control transfer processing for CDC

(2) **C language format**

void R_usb_pcdc_ctrltrans (USB_REQUEST_t *preq, uint16_t ctsq)

(3) **Parameters**

I/O	Parameter	Description
I	USB_REQUEST_t *preg	Pointer to the class request message
I	uint16_t ctsq	Control transfer stage information USB_CS_IDST: Idle or setup stage USB_CS_RDDS: Control read data stage USB_CS_WRDS: Control write data stage USB_CS_WRND: Control write nodata status stage USB_CS_RDSS: Control read status stage USB_CS_WRSS: Control write status stage USB_CS_SQER: Sequence error

(4) **Description**

Register this API function in member ctrltrans of the USB_PCDREG_t structure as the callback function.

This function calls processing for the control transfer stage if the request type is a CDC class request.

(5) **Returned values**

None

Example of Use

```
void pcdc_init(void)
{
        USB_PCDREG_t driver;

        driver.ctrltrans = &R_usb_pcdc_ctrltrans;
        R_usb_pstd_DriverRegistration(&driver);
}
```

3.11 WDTA Control

This LSI provides drivers for controlling the WDTA function.

3.11.1 WDT open

R_WDT_Open

(1) **Synopsis**

WDT open

(2) **C language format**

wdt_err_t R_WDT_Open (uint16_t channel, void * const p_cfg)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	channel	This specifies the WDT channels Setting range: (0)
I	void * const	p_cfg	The pointer that stores the data group to be set in the WDTA-related registers. Timeout period WDT_TIMEOUT_1024 WDT_TIMEOUT_4096 WDT_TIMEOUT_8192 WDT_TIMEOUT_16384 Clock division ratio WDT_CLOCK_DIV_4 WDT_CLOCK_DIV_64 WDT_CLOCK_DIV_128 WDT_CLOCK_DIV_512 WDT_CLOCK_DIV_2048 WDT_CLOCK_DIV_8192 Window stop WDT_WINDOW_END_75 WDT_WINDOW_END_50 WDT_WINDOW_END_25 WDT_WINDOW_END_0 Window start WDT_WINDOW_START_25 WDT_WINDOW_START_50 WDT_WINDOW_START_75 WDT_WINDOW_START_100 Notification of ECM errors WDT_ERROR_ENABLE WDT_ERROR_DISABLE

(4) **Description**

This function initializes WDTA-related registers and sets the options of the WDT counter.

“Figure 8-3 WDT Open Processing” is a flowchart for the WDT Open processing.

(5) **Returned values**

Return value	Meaning
WDT_SUCCESS	IWDT initialized.
WDT_ERR_OPEN_IGNORED	Module already opened.
WDT_ERR_INVALID_ARG	Invalid values included in the element of the p_cfg structure.
WDT_ERR_NULL_PTR	p_cfg pointer null.

Remarks Setting WDT_CFG_PARAM_CHECKING_ENABLE that is defined by r_wdt_config.h to 1 enables checking of the parameters of the arguments.

3.11.2 WDT Control

R_WDT_Control

(1) **Synopsis**

Controlling WDT

(2) **C language format**

wdt_err_t R_WDT_Control(uint16_t channel, wdt_cmd_t const cmd, uint16_t * p_status)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	channel	WDT channel specification Setting range: (0 Only)
I	wdt_cmd_t const	cmd	Specifies the command to be executed IWDT_CMD_GET_STATUS IWDT_CMD_REFRESH_COUNTING
I	uint16_t *	p_status	The pointer to the storage positions of the counter and status flag.

(4) **Description**

This function reads the WDT state of the specified channel and refreshes the down counter of the WDT.

“Figure 8-4 WDT Control Processing” is a flowchart for the WDT Control processing.

(5) **Returned values**

Return value	Meaning
WDT_SUCCESS	Command completed.
WDT_ERR_INVALID_ARG	Argument value invalid.
WDT_ERR_NULL_PTR	p_status null.
WDT_ERR_NOT_OPEND	Open unread.

Remarks Setting WDT_CFG_PARAM_CHECKING_ENABLE that is defined by r_wdt_config.h to 1 enables checking of the parameters of the argument.

3.12 IWDTA Control

This LSI provides drivers for controlling the IWDTA function.

3.12.1 IWDT Open

R_IWDT_Open

(1) **Synopsis**

IWDTA open

(2) **C language format**

iwdt_err_t R_IWDT_Open (void * const p_cfg)

(3) **Parameters**

I/O	Parameter	Description
I	void * const p_cfg	The pointer that stores the data group to be set in the IWDTA-related registers Timeout period IWDT_TIMEOUT_1024 IWDT_TIMEOUT_4096 IWDT_TIMEOUT_8192 IWDT_TIMEOUT_16384 Clock division ratio IWDT_CLOCK __ DIV_1 IWDT_CLOCK __ DIV_16 IWDT_CLOCK __ DIV_32 IWDT_CLOCK __ DIV_64 IWDT_CLOCK_DIV_128 IWDT_CLOCK_DIV_256 Window stop IWDT_WINDOW_END_75 IWDT_WINDOW_END_50 IWDT_WINDOW_END_25 IWDT_WINDOW_END_0 Window start IWDT_WINDOW_START_25 IWDT_WINDOW_START_50 IWDT_WINDOW_START_75 IWDT_WINDOW_START_100 Notification of ECM errors IWDT_ERROR_ENABLE IWDT_ERROR_DISABLE

(4) **Description**

This function initializes IWDTA-related registers and sets the options of the IWDT counter.

“Figure 9-3 IWDT Open Processing” is a flowchart for the IWDTA Open processing.

(5) **Returned values**

Return value	Meaning
IWDT_SUCCESS	IWDT initialized.
IWDT_ERR_OPEN_IGNORED	Module already opened.
IWDT_ERR_INVALID_ARG	Invalid values included in the element of the p_cfg structure.
IWDT_ERR_NULL_PTR	p_cfg pointer null.

Remarks Setting IWDT_CFG_PARAM_CHECKING_ENABLE that is defined by r_iwdt_config.h to 1 enables checking of the parameters of the arguments.

3.12.2 IWDT Control

R_IWDT_Control

(1) **Synopsis**

Controlling IWDTA

(2) **C language format**

iwdt_err_t R_IWDT_Control(iwdt_cmd_t const cmd, uint16_t * p_status)

(3) **Parameters**

I/O	Parameter		Description
I	iwdt_cmd_t const	cmd	Specifies the command to be executed. IWDT_CMD_GET_STATUS IWDT_CMD_REFRESH_COUNTING
I	uint16_t *	p_status	The pointer to the storage positions of the counter and status flag

(4) **Description**

This function reads the state of the IWDT and refreshes the down counter of the IWDT.

“Figure 9-4 IWDT Control Processing” is a flowchart for the IWDTA Control processing.

(5) **Returned values**

Return value	Meaning
IWDT_SUCCESS	Command completed.
IWDT_ERR_INVALID_ARG	Argument value invalid.
IWDT_ERR_NULL_PTR	p_status null.
IWDT_ERR_NOT_OPEND	Open unread.

Remarks Setting IWDT_CFG_PARAM_CHECKING_ENABLE that is defined by r_iwdt_config.h to 1 enables checking of the parameters of the argument.

3.13 RIIC Control

This LSI provides drivers for controlling the RIIC function.

3.13.1 RIIC Open

R_RIIC_Open

(1) **Synopsis**

This function is required first when using this module.

(2) **C language format**

riic_return_t R_RIIC_Open(riic_info_t * p_riic_info)

(3) **Parameters**

I/O	Parameter	Description
I	riic_info_t * p_riic_info	This is the pointer to the I2C communication information structure. Only the members of this structure that are used by this function are shown below. riic_ch_dev_status_t dev_sts; To be updated by calling the driver. For details, see Figure 10-6 uint8_t ch_no;

(4) **Description**

This function makes initial settings of the RIIC to start communications. It sets the RIIC channel specified by the argument. If the state of the channel is "uninitialized (RIIC_NO_INIT)", the following processes are performed:

- Setting the state flag
- Setting I/O ports
- Allocating I2C output ports
- Releasing the stopped state of the RIIC module
- Initializing the variables used by the API
- Initializing the RIIC registers used for RIIC communications
- Disabling the RIIC interrupts

(5) **Returned values**

Return value	Meaning
RIIC_SUCCESS	Processing completed successfully.
RIIC_ERR_LOCK_FUNC	The API is locked by another task.
RIIC_ERR_INVALID_CHAN	Nonexistent channel.
RIIC_ERR_INVALID_ARG	Invalid argument.
RIIC_ERR_OTHER	An invalid event occurred in the current state.

Example

```
volatile riic_return_t ret;
riic_info_t iic_info_m;

iic_info_m.dev_sts = 0;
iic_info_m.ch_no = 0;

ret = R_RIIC_Open(&iic_info_m);
```

3.13.2 RIIC Master Send

R_RIIC_MasterSend

(1) **Synopsis**

This function is used when this module starts transmission as a master device.

(2) **C language format**

riic_return_t R_RIIC_MasterSend(riic_info_t * p_riic_info)

(3) **Parameters**

I/O	Parameter	Description
I	riic_info_t * p_riic_info	This is the pointer to the I2C communication information structure. Only the members of this structure that are used by this function are shown below. riic_ch_dev_status_t dev_sts; — To be updated by calling the driver. For details, see Figure 10-6 uint8_t ch_no; riic_callback callbackfunc; uint32_t cnt2nd; — To be updated for only transmission patterns 1 and 2 uint32_t cnt1st; — To be updated for only transmission pattern 1 uint8_t * p_data2nd; uint8_t * p_data1st; uint8_t * p_slv_adr;

(4) **Description**

This function starts the RIIC master transmission. The transmission is performed with the RIIC channel and transmission pattern specified by the arguments. If the state of the channel is "idle" (RIIC_IDLE, RIIC_FINISH, or RIIC_NACK), the following processes are performed:

- Setting the state flag
- Initializing the variables used by the API
- Enabling the RIIC interrupts
- Generating a start condition

The transmission pattern can be selected from four patterns by the argument settings.

See Table 3-78 for the specification method and allowable argument settings for each transmission pattern. See Figure 3-3 to Figure 3-6 for the signal waveforms of each transmission pattern.

When setting the slave address, store it without shifting 1 bit to left.

(5) **Returned values**

Return value	Meaning
RIIC_SUCCESS	Processing completed successfully.
RIIC_ERR_INVALID_CHAN	Nonexistent channel.
RIIC_ERR_INVALID_ARG	Invalid argument.
RIIC_ERR_NO_INIT	Uninitialized state.
RIIC_ERR_BUS_BUSY	Bus busy.
RIIC_ERR_AL	Arbitration-lost error occurred.
RIIC_ERR_OTHER	An invalid event occurred in the current state.

Example

```
/* for MasterSend(Pattern 1) */
#include "r_riic_rx_if.h"

void CallbackMaster(void);
void main(void);

void main(void)
{
    volatile riic_return_t ret;
    riic_info_t iic_info_m;
    uint8_t addr_eeprom[1]={0x50};
    uint8_t access_addr1[1]={0x00};
    uint8_t mst_send_data[5]={0x81,0x82,0x83,0x84,0x85};

    /* Sets IIC Information for sending pattern 1. */
    iic_info_m.dev_sts = 0;
    iic_info_m.ch_no = 0;
    iic_info_m.callbackfunc = &CallbackMaster;
    iic_info_m.cnt2nd = 3;
    iic_info_m.cnt1st = 1;
    iic_info_m.p_data2nd = mst_send_data;
    iic_info_m.p_data1st = access_addr1;
    iic_info_m.p_slv_adr = addr_eeprom;

    /* RIIC open */
    ret = R_RIIC_Open(&iic_info_m);

    /* RIIC send start */
    ret = R_RIIC_MasterSend(&iic_info_m);
    while(1);
}

void CallbackMaster(void)
{
    /* callback process */
}
```

Special Notes:

This function is used when this module starts transmission as a master device.

Table 3-78 Allowable Argument Settings for Each Transmission Pattern

Structure Member	User Settable Range			
	Master Transmission Pattern 1	Master Transmission Pattern 2	Master Transmission Pattern 3	Master Transmission Pattern 4
*p_slv_adr	Pointer to the slave address storage buffer	Pointer to the slave address storage buffer	Pointer to the slave address storage buffer	FIT_NO_PTR*1
*p_data1st	Pointer to the first data storage buffer for transmission	FIT_NO_PTR*1	FIT_NO_PTR*1	FIT_NO_PTR*1
*p_data2nd	Pointer to the second data storage buffer for transmission	Pointer to the second data storage buffer for transmission	FIT_NO_PTR*1	FIT_NO_PTR*1
cnt1st	0000 0001h to FFFF FFFFh *2	0	0	0
cnt2nd	0000 0001h to FFFF FFFFh *2	0000 0001h to FFFF FFFFh *2	0	0
callbackfunc	Specify the function name to be used.	Specify the function name to be used.	Specify the function name to be used.	Specify the function name to be used.
ch_no	00h to FFh	00h to FFh	00h to FFh	00h to FFh
dev_sts	Device state flag	Device state flag	Device state flag	Device state flag
rsv1,rsv2	Reserved (value set here has no effect)	Reserved (value set here has no effect)	Reserved (value set here has no effect)	Reserved (value set here has no effect)

Note 1. When using pattern 2, 3, or 4, set "FIT_NO_PTR" for the applicable structure members as shown in the table above.

Note 2. 0 cannot be set.

(a) Pattern 1

As a master device, this function transmits data in two buffers (for the first data and second data) to the slave device.

A start condition (ST) is generated first and then the slave device address is transmitted. The eighth bit specifies the transfer direction, and so this bit is set to 0 (write) when transmitting data. Then, the first data is transmitted. The first data is used when there is data to be transmitted before performing the data transmission. For example, if the slave device is an EEPROM, an internal address in the EEPROM can be transmitted. Next, the second data is transmitted. The second data is the data to be written to the slave device. When a data transmission has started and all data transmissions have completed, a stop condition (SP) is generated, and the bus is released.

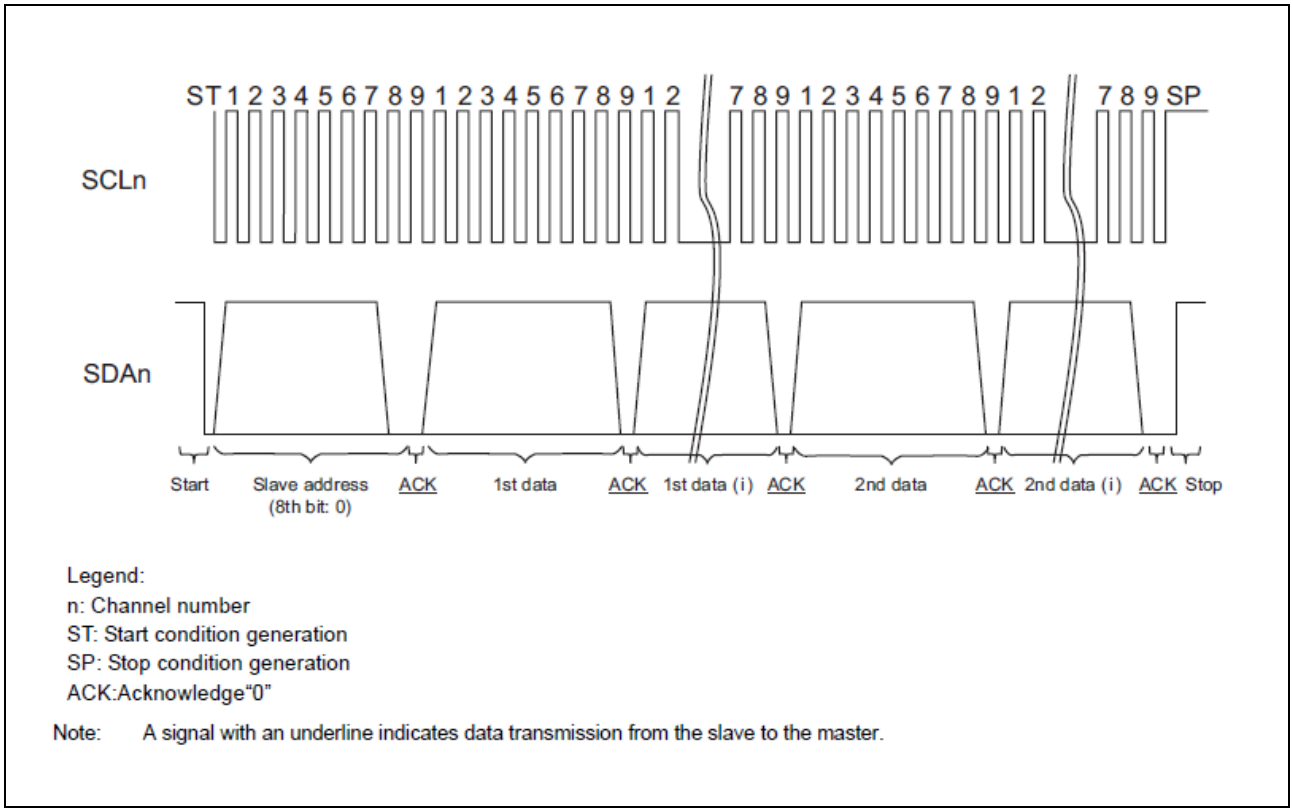
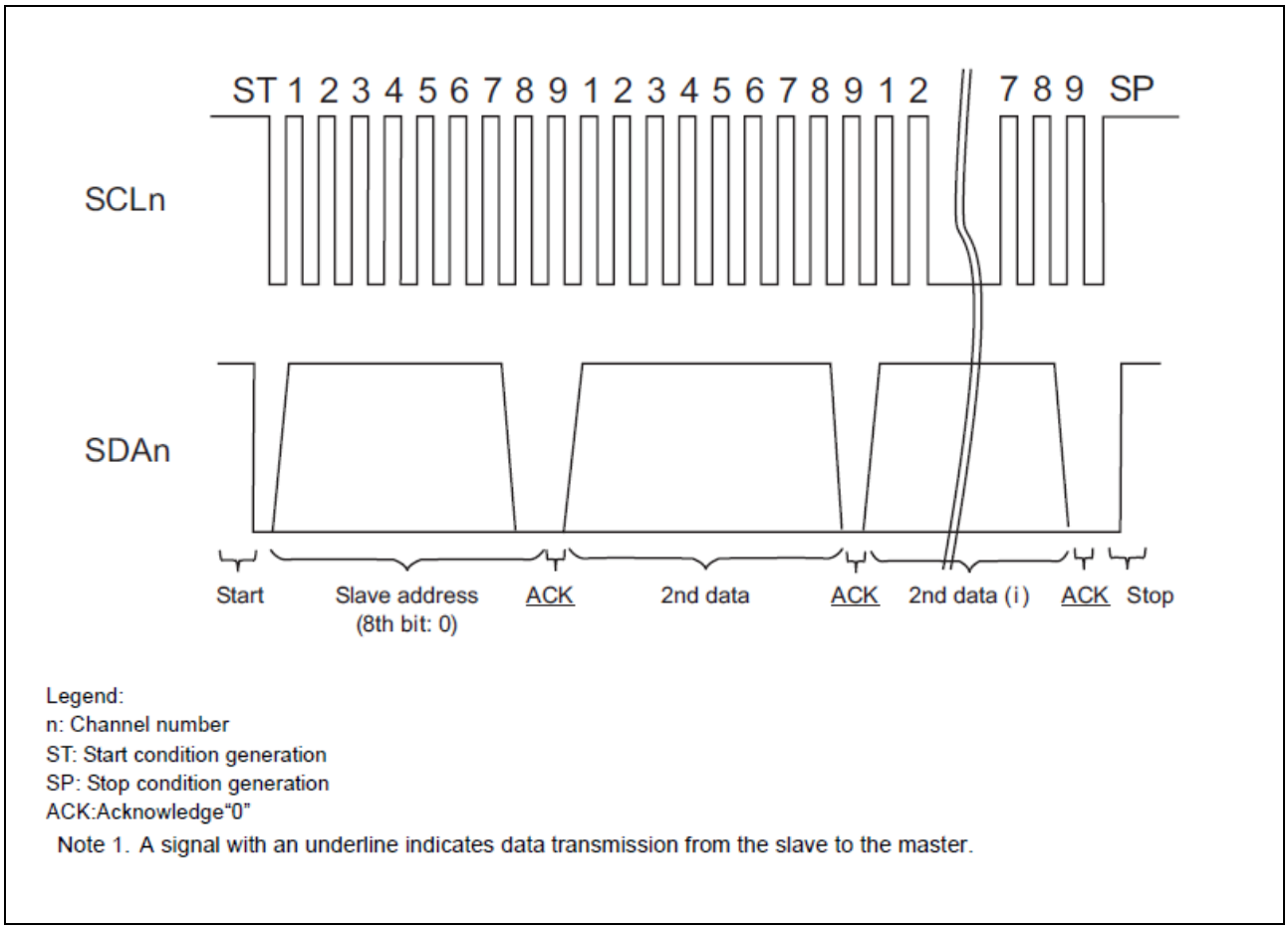


Figure 3-3 Signals for Master Transmission Pattern 1

(b) **Pattern 2**

As a master device, this function transmits data in a buffer (for the second data) to the slave device.

Operations from start condition (ST) generation through to slave device address transmission are the same as for pattern 1. Then the second data is transmitted without transmitting the first data. When all data transmissions have completed, a stop condition (SP) is generated and the bus is released.



Legend:
n: Channel number
ST: Start condition generation
SP: Stop condition generation
ACK:Acknowledge“0”
Note 1. A signal with an underline indicates data transmission from the slave to the master.

Figure 3-4 Signals for Master Transmission Pattern 2

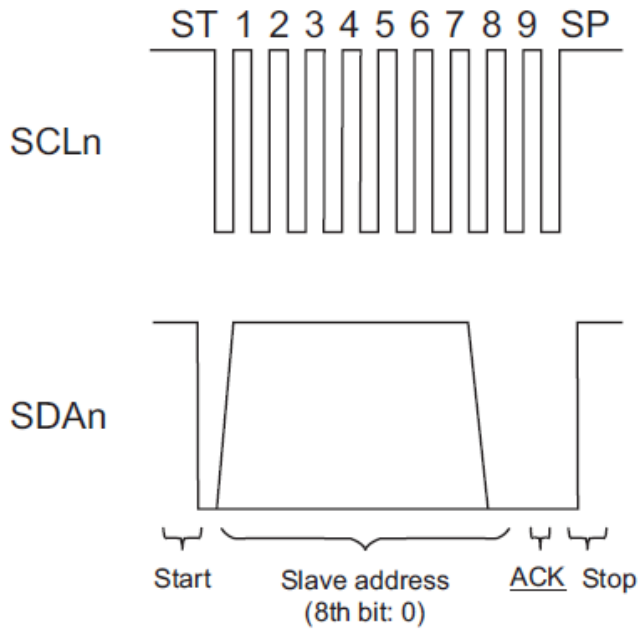
(c) **Pattern 3**

As a master device, this function transmits only the slave address to the slave device.

Operations from start condition (ST) generation through to slave address transmission are the same as for pattern 1.

After transmitting the slave address, if neither the first data nor the second data is set, data transmission is not performed, then a stop condition (SP) is generated, and the bus is released.

This pattern is useful for detecting connected devices or when performing acknowledge polling to verify the EEPROM rewriting state.



Legend:
n: Channel number
ST: Start condition generation
SP: Stop condition generation
ACK:Acknowledge"0"
Note 1. A signal with an underline indicates data transmission from the slave to the master.

Figure 3-5 Signals for Master Transmission Pattern 3

(d) **Pattern 4**

As a master device, this function transmits only a start condition and stop condition to the slave device.

After a start condition (ST) is generated, if the slave address, first data, and second data are not set, slave address transmission and data transmission are not performed, then a stop condition (SP) is generated and the bus is released. This pattern is useful for just releasing the bus.

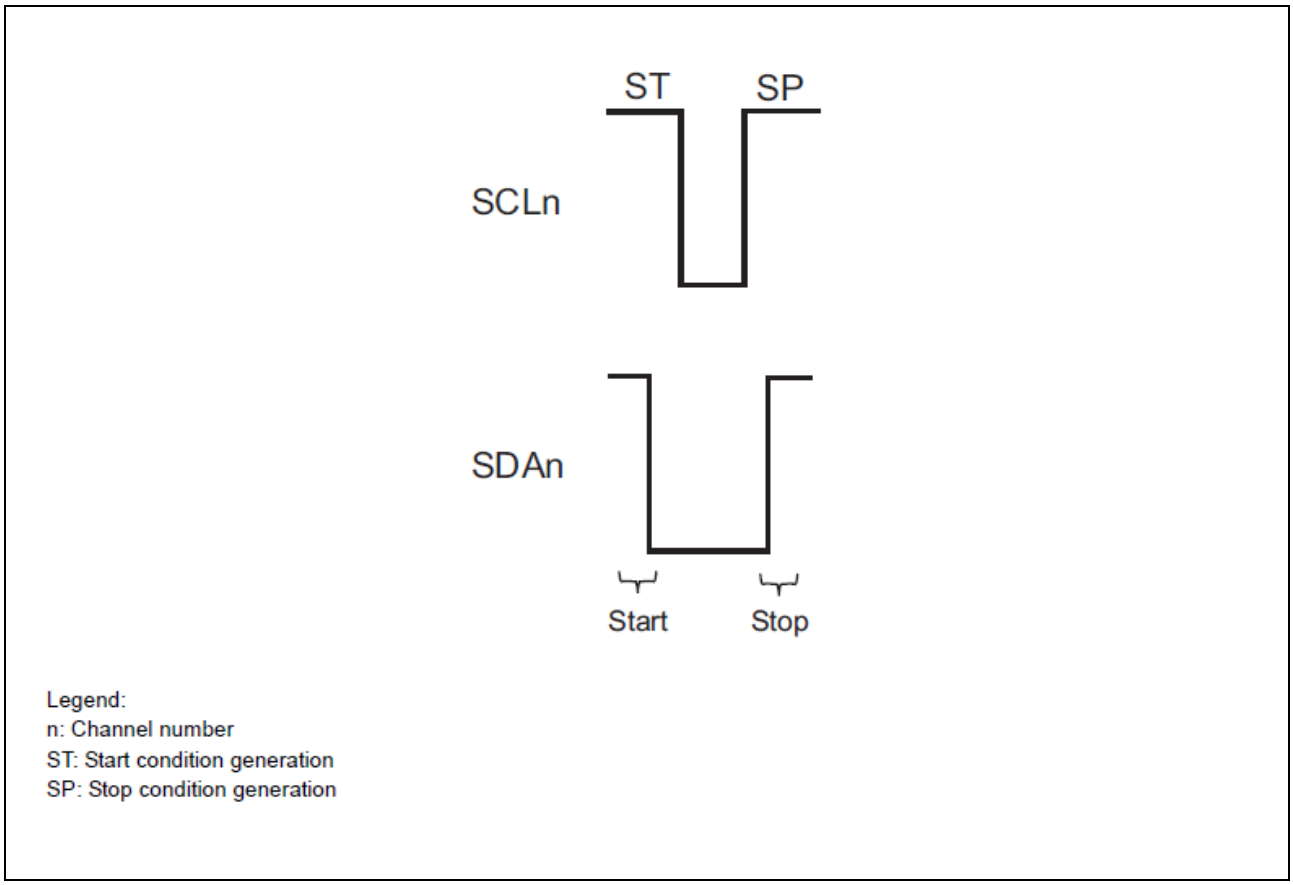


Figure 3-6 Signals for Master Transmission Pattern 4

3.13.3 RIIC Master Receive

R_RIIC_MasterReceive

(1) **Synopsis**

This function is used when the module starts reception as a master device.

(2) **C language format**

riic_return_t R_RIIC_MasterReceive(riic_info_t * p_riic_info)

(3) **Parameters**

I/O	Parameter	Description
I	riic_info_t * p_riic_info	This is the pointer to the I2C communication information structure. Only the members of this structure that are used by this function are shown below. riic_ch_dev_status_t dev_sts; — To be updated by calling the driver. For details, see Figure 10-6 uint8_t ch_no; riic_callback callbackfunc; uint32_t cnt2nd; — To be updated. uint32_t cnt1st; — To be updated only for master composite. uint8_t * p_data2nd; uint8_t * p_data1st; uint8_t * p_slv_adr;

(4) **Description**

This function starts the RIIC master reception. The reception is performed with the RIIC channel and reception pattern specified by the arguments. If the state of the channel is "idle" (RIIC_IDLE, RIIC_FINISH, or RIIC_NACK), the following processes are performed:

- Setting the state flag
- Initializing the variables used by the API
- Enabling the RIIC interrupts
- Generating a start condition

The reception pattern can be selected from master reception and master composite by the argument settings.

See Table 3-79 for the specification method and allowable argument settings for each reception pattern. See Figure 3-7 and Figure 3-8 for the signal waveforms of each reception pattern.

When setting the slave address, store it without shifting 1 bit to left.

(5) **Returned values**

Return value	Meaning
RIIC_SUCCESS	Processing completed successfully.
RIIC_ERR_INVALID_CHAN	Nonexistent channel.
RIIC_ERR_INVALID_ARG	Invalid argument.
RIIC_ERR_NO_INIT	Uninitialized state.
RIIC_ERR_BUS_BUSY	Bus busy.
RIIC_ERR_AL	Arbitration-lost error occurred.
RIIC_ERR_OTHER	An invalid event occurred in the current state.

Example

```
/* for MasterReceive(combination mode) */
#include "r_riic_rx_if.h"

void CallbackMaster(void);
void main(void);

void main(void)
{
        volatile riic_return_t ret;
        riic_info_t iic_info_m;
        uint8_t addr_eeprom[1]={0x50};
        uint8_t access_addr1[1]={0x00};
        uint8_t mst_store_area[5]={0xFF,0xFF,0xFF,0xFF,0xFF};

        /* Sets IIC Information. */
        iic_info_m.dev_sts = 0;
        iic_info_m.ch_no = 0;
        iic_info_m.callbackfunc = &CallbackMaster;
        iic_info_m.cnt2nd = 3;
        iic_info_m.cnt1st = 1;
        iic_info_m.p_data2nd = mst_store_area;
        iic_info_m.p_data1st = access_addr1;
        iic_info_m.p_slv_adr = addr_eeprom;

        /* RIIC open */
        ret = R_RIIC_Open(&iic_info_m);

        /* RIIC receive start */
        ret = R_RIIC_MasterReceive(&iic_info_m);

        while(1);
}

void CallbackMaster(void)
{
        /* callback process */
}
```

Special Notes

The following table lists the allowable argument settings for each reception pattern.

Table 3-79 Allowable Argument Settings for Each Reception Pattern

Structure Member	User Settable Range	
	Master Reception	Master Composite
*p_slv_adr	Pointer to the slave address storage buffer	Pointer to the slave address storage buffer
*p_data1st	Not used (value set here has no effect)	Pointer to the first data storage buffer for transmission
*p_data2nd	Pointer to the second data storage buffer for reception	Pointer to the second data storage buffer for reception
dev_sts	Device state flag	Device state flag
cnt1st[*1]	0	0000 0001h to FFFF FFFFh[*2]
cnt2nd	0000 0001h to FFFF FFFFh[*2]	0000 0001h to FFFF FFFFh[*2]
callbackfunc	Specify the function name to be used.	Specify the function name to be used.
ch_no	00h to FFh	00h to FFh
rsv1,rsv2	Reserved (value set here has no effect)	Reserved (value set here has no effect)

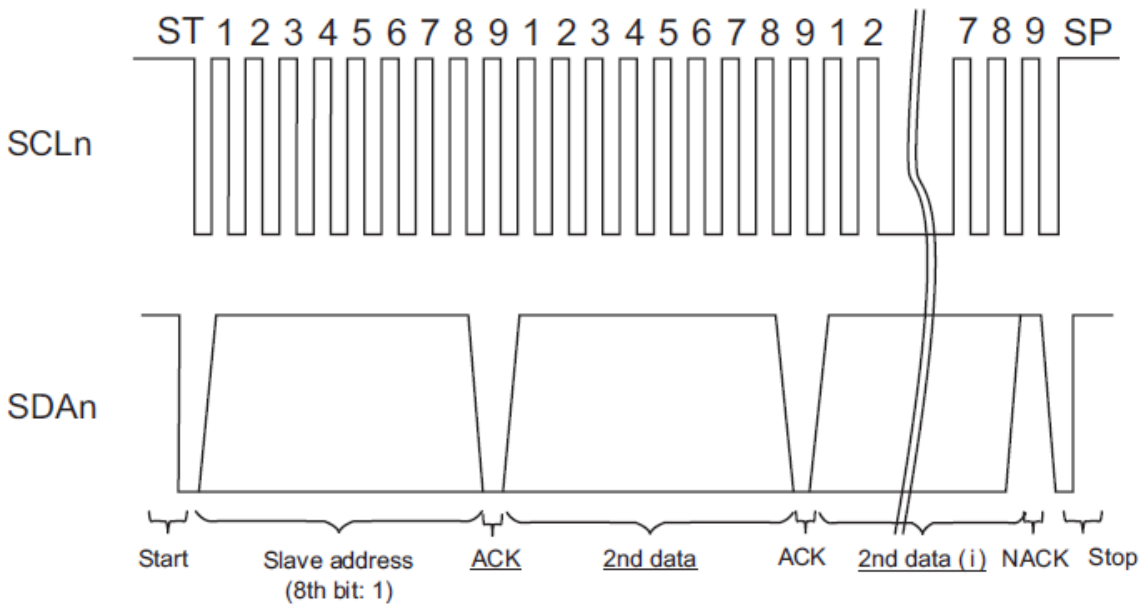
Note 1. The reception pattern is determined by whether the first data is 0 or not.

Note 2. 0 cannot be set.

(a) Master Reception

As a master device, this function receives data from the slave device.

A start condition (ST) is generated first and then the slave device address is transmitted. The eighth bit specifies the transfer direction, and so this bit is set to 1 (read) when receiving data. Then, data reception starts. An ACK is sent each time one byte of data is received, but when the last data is received, a NACK is sent to notify the slave device that all data receptions have completed. When all data receptions have completed, a stop condition (SP) is generated and the bus is released.



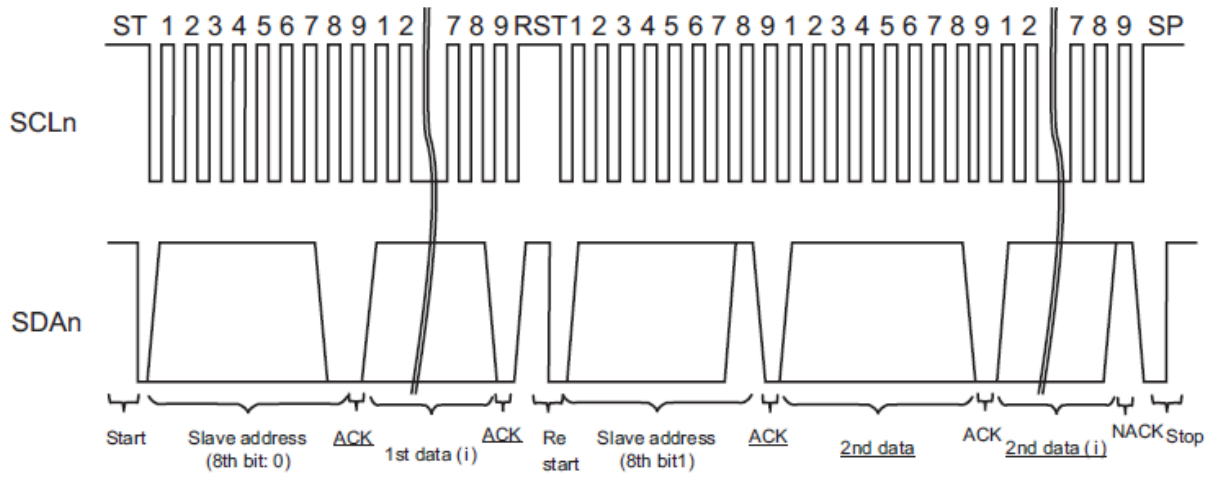
Legend:
n: Channel number
ST: Start condition generation
SP: Stop condition generation
NACK:Acknowledge“1”
ACK:Acknowledge“0”
Note 1. A signal with an underline indicates data transmission from the slave to the master.

Figure 3-7 Signals for Master Reception

(b) Master Composite

As a master device, this function transmits data to the slave device. After the transmission completes, this function generates a restart condition and receives data from the slave.

A start condition (ST) is generated first and then the slave device address is transmitted. The eighth bit specifies the transfer direction, and so this bit is set to 0 (write) when transmitting data. Next, the first data is transmitted. When the data transmission completes, a restart condition (RST) is generated and the slave address is transmitted. Then, the eighth bit is set to 1 (read) and a data reception starts. An ACK is sent each time one byte of data is received, but when the last data is received, a NACK is sent to notify the slave device that all data receptions have completed. When all data receptions have completed, a stop condition (SP) is generated and the bus is released.



Legend:
n: Channel number
ST: Start condition generation
SP: Stop condition generation
RST: Restart condition generation
NACK:Acknowledge“1”
ACK:Acknowledge“0”
Note 1. A signal with an underline indicates data transmission from the slave to the master.

Figure 3-8 Signals for Master Composite

3.13.4 RIIC Slave Transfer

R_RIIC_SlaveTransfer

(1) Synopsis

This function is used when the module functions as a slave device to prepare for transmission and reception.

(2) C language format

riic_return_t R_RIIC_SlaveTransfer(riic_info_t * p_riic_info)

(3) Parameters

I/O	Parameter		Description	
I	riic_info_t *	p_riic_info	This is the pointer to the I2C communication information structure. Only the members of this structure that are used by this function are shown below.	
			riic_ch_dev_status_t dev_sts;	To be updated by calling the driver. For details, see Figure 10-6
			uint8_t ch_no;	
			riic_callback callbackfunc;	
			uint32_t cnt2nd;	To be updated for only slave reception.
			uint32_t cnt1st;	To be updated for only slave transmission.
			uint8_t * p_data2nd;	
			uint8_t * p_data1st;	
			uint8_t * p_slv_adr;	

(4) Description

This function prepares for the RIIC slave transmission or slave reception. If this function is called while the master is communicating, an error occurs. This function sets the RIIC channel specified by the argument. If the state of the channel is "idle" (RIIC_IDLE, RIIC_FINISH, or RIIC_NACK), the following processes are performed:

- Setting the state flag
- Initializing the variables used by the API
- Initializing the RIIC registers used for RIIC communications
- Enabling the RIIC interrupts
- Setting the slave address and enabling the slave address match interrupt

The operation can be selected from preparation for slave reception, slave transmission, or both of them by the argument settings.

See Table 3-80 for the allowable argument settings for each slave operation pattern.

See Figure 3-9 for the signal waveforms of the reception pattern and Figure 3-10 for the signal waveforms of the transmission pattern.

(5) **Returned values**

Return value	Meaning
RIIC_SUCCESS	Processing completed successfully.
RIIC_ERR_INVALID_CHAN	Nonexistent channel.
RIIC_ERR_INVALID_ARG	Invalid argument.
RIIC_ERR_NO_INIT	Uninitialized state.
RIIC_ERR_BUS_BUSY	Bus busy.
RIIC_ERR_AL	Arbitration-lost error occurred.
RIIC_ERR_OTHER	An invalid event occurred in the current state.

Example

```
/* for MasterReceive(combination mode) */
#include "r_riic_rx_if.h"

void CallbackMaster(void);
void CallbackSlave(void);
void main(void);

void main(void)
{
   volatile riic_return_t ret;
   riic_info_t iic_info_m;
   riic_info_t iic_info_s;
   uint8_t addr_eeprom[1]={0x50};
   uint8_t access_addr1[1]={0x00};
   uint8_t mst_send_data[5]={0x81,0x82,0x83,0x84,0x85};
   uint8_t slv_send_data[5]={0x71,0x72,0x73,0x74,0x75};
   uint8_t mst_store_area[5]={0xFF,0xFF,0xFF,0xFF,0xFF};
   uint8_t slv_store_area[5]={0xFF,0xFF,0xFF,0xFF,0xFF};

   /* Sets IIC Information for Master Send. */
   iic_info_m.dev_sts = 0;
   iic_info_m.ch_no = 0;
   iic_info_m.callbackfunc = &CallbackMaster;
   iic_info_m.cnt2nd = 3;
   iic_info_m.cnt1st = 1;
   iic_info_m.p_data2nd = mst_store_area;
   iic_info_m.p_data1st = access_addr1;
   iic_info_m.p_slv_adr = addr_eeprom;

   /* Sets IIC Information for Slave Transfer. */
   iic_info_s.dev_sts = 0;
   iic_info_s.ch_no = 0;
   iic_info_s.callbackfunc = &CallbackSlave;
   iic_info_s.cnt2nd = 3;
   iic_info_s.cnt1st = 3;
   iic_info_s.p_data2nd = slv_store_area;
   iic_info_s.p_data1st = slv_send_data;
   iic_info_s.p_slv_adr = (uint8_t*)FIT_NO_PTR;

   /* RIIC open */
   ret = R_RIIC_Open(&iic_info_m);

   /* RIIC slave transfer enable */
   ret = R_RIIC_SlaveTransfer(&iic_info_s);

   /* RIIC master send start */
   ret = R_RIIC_MasterSend(&iic_info_m);
   while(1);
}
void CallbackMaster(void)
{
   /* callback process (master)*/
}
void CallbackSlave(void)
{
   /* callback process (slave)
}
```

Special Notes

The following table lists the allowable argument settings for each slave operation pattern.

Table 3-80 Allowable Argument Settings for Each Reception Pattern

Structure Member	User Settable Range	
	Slave Reception	Slave Transmission
*p_slv_adr	Not used (value set here has no effect)	Not used (value set here has no effect)
*p_data1st	(For slave transmission)	Pointer to the first data storage buffer for transmission*1
*p_data2nd	Pointer to the second data storage buffer for reception*2	(For slave reception)
dev_sts	Device state flag	Device state flag
cnt1st	(For slave transmission)	0000 0001h to FFFF FFFFh
cnt2nd	0000 0001h to FFFF FFFFh	(For slave reception)
callbackfunc	Specify the function name to be used.	Specify the function name to be used.
ch_no	00h to FFh	00h to FFh
rsv1,rsv2	Reserved (value set here has no effect)	Reserved (value set here has no effect)

Note 1. Set this when performing slave transmission.

When slave transmission is not used, set FIT_NO_PTR.

Note 2. Set this when performing slave reception.

When slave reception is not used, set FIT_NO_PTR.

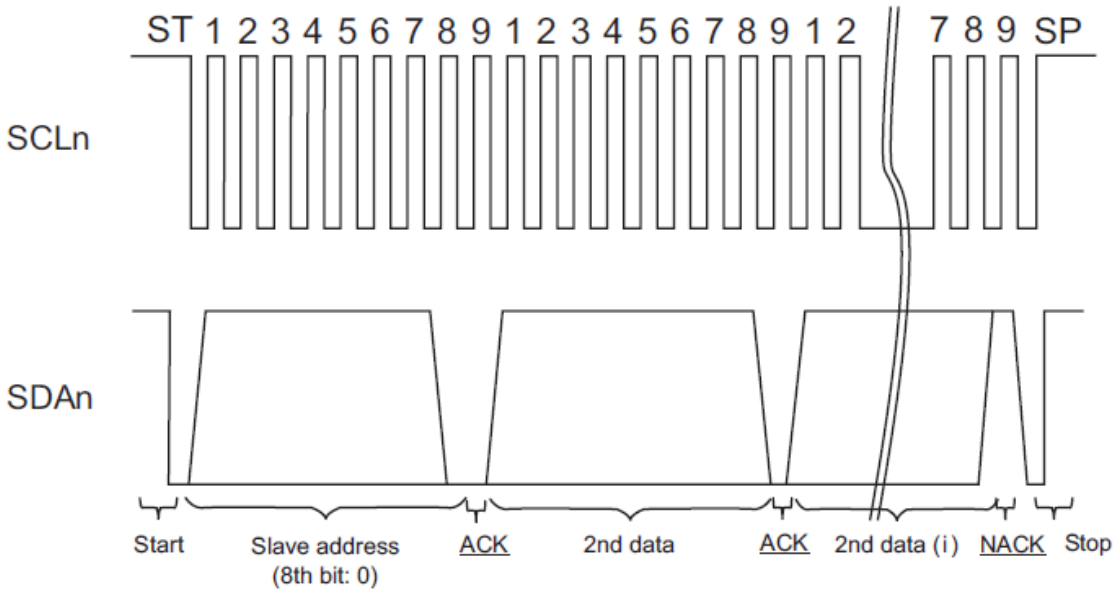
(a) Slave Reception

As a slave device, this function receives data from the master device.

When the slave address specified by the master device matches the slave address specified in r_riic_config_if.h, slave transmission and reception starts. This module performs processing by automatically determining whether the operation is slave reception or slave transmission according to the eighth bit (transfer direction specification bit) of the slave address.

After a start condition (ST) generated by the master device is detected, if the received slave address matches its own address and the eighth bit of the slave address is 0 (write), then the module starts reception operation as a slave device .

When the last data (the number of received data items specified in the RIIC communication information structure) is received, a NACK is returned to the master device to notify that all the necessary data has been received.



Legend:
n: Channel number
ST: Start condition generation
SP: Stop condition generation
NACK:Acknowledge“1”
ACK:Acknowledge“0”
Note 1. A signal with an underline indicates data transmission from the slave to the master.

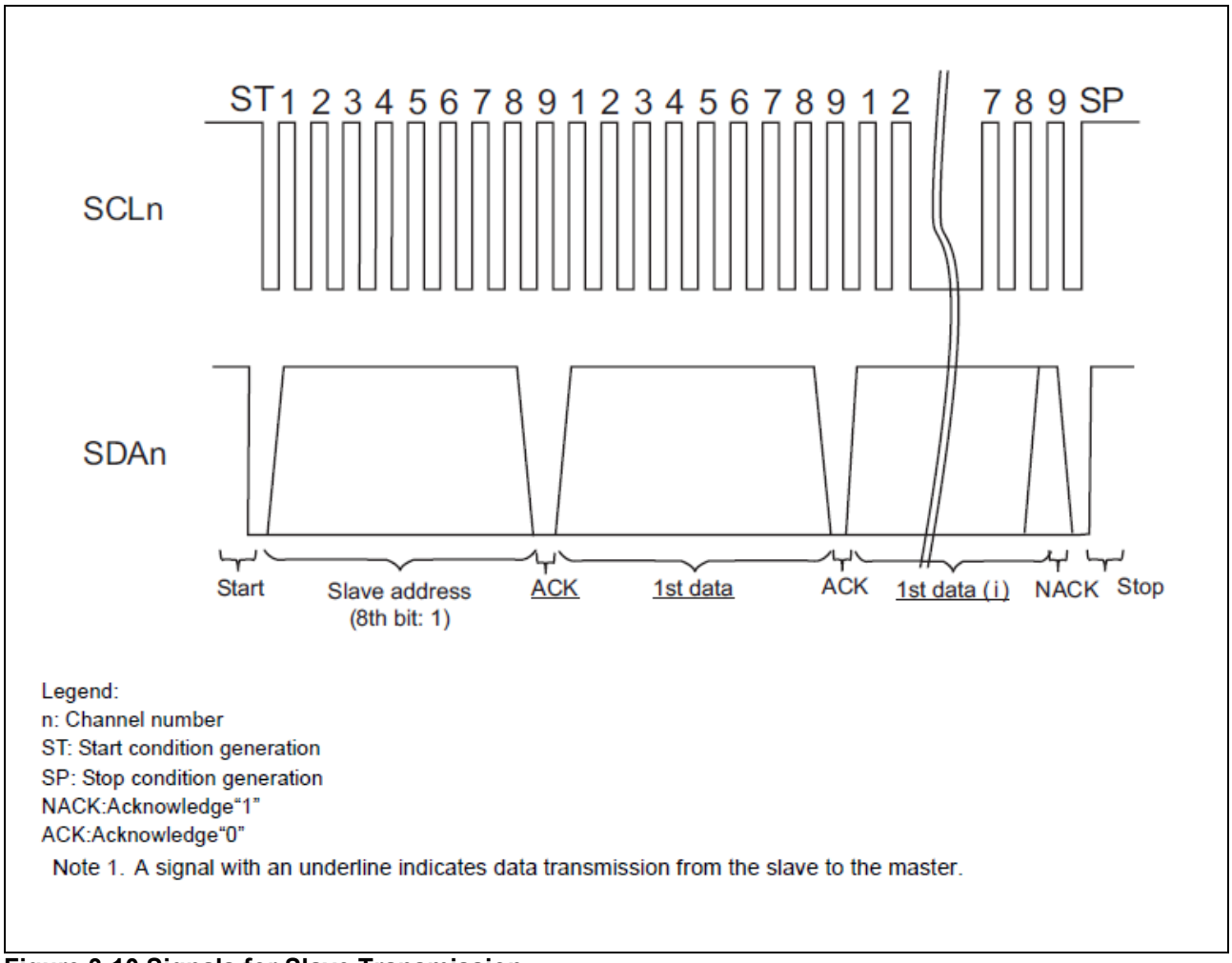
Figure 3-9 Signals for Slave Reception

(b) Slave Transmission

As a slave device, this function transmits data to the master device.

When the slave address specified by the master device matches the slave address specified in r_riic_config_if.h, slave transmission and reception starts. This module performs processing by automatically determining whether the operation is slave reception or slave transmission according to the eighth bit (transfer direction specification bit) of the slave address.

After a start condition (ST) from the master device is detected, if the received slave address matches its own address and the eighth bit of the slave address is 1 (read), then the module starts transmission operation as a slave device. If the transmission request exceeds the number of sent data items specified in the I2C communication information structure, 0xFF is sent as data. The slave continues transmitting data until a stop condition (SP) is detected.



Legend:
n: Channel number
ST: Start condition generation
SP: Stop condition generation
NACK:Acknowledge“1”
ACK:Acknowledge“0”
Note 1. A signal with an underline indicates data transmission from the slave to the master.

Figure 3-10 Signals for Slave Transmission

3.13.5 RIIC Get Status

R_RIIC_GetStatus

(1) **Synopsis**

This function is used when verifying the state of this module.

(2) **C language format**

riic_sts_flg_t R_RIIC_GetStatus(riic_info_t * p_riic_info, riic_mcu_status_t * p_riic_status)

(3) **Parameters**

I/O	Parameter		Description
I	riic_info_t *	p_riic_info	This is the pointer to the I2C communication information structure. Only the members of this structure that are used by this function are shown below. riic_ch_dev_status_t dev_sts; To be updated by calling the driver. uint8_t ch_no; For details, see Figure 10-6
I	riic_mcu_status_t *	p_riic_status	Pointer to the variable to store the RIIC state.

(4) **Description**

This function returns the state of this module.

This function obtains the state of the RIIC channel specified in the argument by reading the registers, pin levels, variables, and others, and then returns the state as a return value (32-bit structure).

When this function is called, the RIIC arbitration-lost detection flag and NACK flag are cleared to 0. If the device state is "RIIC_ AL", the value is updated to "RIIC_FINISH".

(5) **Returned values**

Return value	Meaning
RIIC_SUCCESS	Processing completed successfully.
RIIC_ERR_INVALID_CHAN	Nonexistent channel.
RIIC_ERR_INVALID_ARG	Invalid argument.

Example

```
volatile riic_return_t ret;
riic_info_t iic_info_m;
riic_mcu_status_t riic_status;

iic_info_m.ch_no = 0;

ret = R_RIIC_GetStatus(&iic_info_m, &riic_status);
```

3.13.6 RIIC Control

R_RIIC_Control

(1) Synopsis

This function is mainly used when a communication error occurs.

It outputs conditions, high impedance signals to the SDA pin, and one-shot of the SCL clock. It also resets the RIIC module.

(2) C language format

riic_return_t R_RIIC_Control(r_riic_info_t *p_riic_info, uint8_t ctrl_ptn)

(3) Parameters

I/O	Parameter	Description
I	riic_info_t p_riic_info *	This is the pointer to the I2C communication information structure. Only the members of this structure that are used by this function are shown below. riic_ch_dev_status_t dev_sts; — To be updated when "RIIC_GEN_RESET" is specified as the output pattern. For details, see Figure 10-6 uint8_t ch_no;
I	uint8_t ctrl_ptn	Specifies the output pattern.

(4) Description

This function outputs control signals for the RIIC module. It outputs conditions specified by the arguments, high impedance signals to the SDA pin, and one-shot of the SCL clock. It also resets the RIIC module.

See Table 3-56 for the values that can be specified as output patterns.

The following output patterns can be specified simultaneously. When specifying multiple patterns simultaneously, separate them with a vertical bar ("|").

- The following output patterns can be specified simultaneously by combining two or three of them: RIIC_GEN_START_CON, RIIC_GEN_STOP_CON, and RIIC_GEN_RESTART_CON
- The following two can specified simultaneously: RIIC_GEN_SDA_HI_Z and RIIC_GEN_SCL_ONESHOT

```
RIIC_GEN_START_CON      0x01 /* Start condition generation */
RIIC_GEN_STOP_CON       0x02 /* Stop condition generation */
RIIC_GEN_RESTART_CON    0x04 /* Restart condition generation */
RIIC_GEN_SDA_HI_Z       0x08 /* SDA pin set to high impedance */
RIIC_GEN_SCL_ONESHOT    0x10 /* SCL clock one-shot output */
RIIC_GEN_RESET          0x20 /* RIIC module reset */
```

(5) **Returned values**

Return value	Meaning
RIIC_SUCCESS	Processing completed successfully.
RIIC_ERR_INVALID_CHAN	Nonexistent channel.
RIIC_ERR_INVALID_ARG	Invalid argument.
RIIC_ERR_NO_INIT	Uninitialized state.
RIIC_ERR_BUS_BUSY	Bus busy.
RIIC_ERR_AL	Arbitration-lost error occurred.
RIIC_ERR_OTHER	An invalid event occurred in the current state.

Example

```
/* Outputs an extra SCL clock cycle after changes the SDA pin in a high-impedance state*/
volatile riic_return_t ret;
riic_info_t iic_info_m;

iic_info_m.ch_no = 0;

ret = R_RIIC_Control(&iic_info_m, RIIC_GEN_SDA_HI_Z | RIIC_GEN_SCL_ONESHOT);
```

3.13.7 RIIC Close

R_RIIC_Close

(1) **Synopsis**

This function is used when completing the RIIC communication and releasing the RIIC module used.

(2) **C language format**

riic_return_t R_RIIC_Close(riic_info_t *p_riic_info)

(3) **Parameters**

I/O	Parameter	Description
I	riic_info_t * p_riic_info	This is the pointer to the I2C communication information structure. Only the members of this structure that are used by this function are shown below. riic_ch_dev_status_t dev_sts; To be updated by calling the driver. For details, see Figure 10-6 uint8_t ch_no;

(4) **Description**

This function configures the settings to complete the RIIC communication. It disables the RIIC channel specified by the argument. This function performs the following processes:

- Changing the RIIC module to a stopped state
- Releasing the RIIC output ports (switching SCL0 and SDA0 to port mode (input))
- Disabling the RIIC interrupts

To restart the communication, call the R_RIIC_Open() function (initialization function). If the communication is forcibly terminated, that communication is not guaranteed.

(5) **Returned values**

Return value	Meaning
RIIC_SUCCESS	Processing completed successfully.
RIIC_ERR_INVALID_CHAN	Nonexistent channel.
RIIC_ERR_INVALID_ARG	Invalid argument.

Example

```
volatile riic_return_t ret;
riic_info_t iic_info_m;
iic_info_m.ch_no = 0;
ret = R_RIIC_Close(&iic_info_m);
```

3.13.8 RIIC Get Version

R_RIIC_GetVersion

(1) **Synopsis**

This function returns the API version.

(2) **C language format**

uint32_t R_RIIC_GetVersion(void)

(3) **Parameters**

None

(4) **Description**

This function returns the version number of this API.

(5) **Returned values**

Return value	Meaning
-	Version number

Example

```
uint32_t version;

version = R_RIIC_GetVersion();
```

3.14 USB host Control

Provides a driver for controlling the USB host.

3.14.1 Data transfer request

R_usb_hstd_TransferStart

(1) **Synopsis**

Data transfer execution request

(2) **C language format**

USB_ER_t R_usb_hstd_TransferStart(USB_UTR_t * utr)

(3) **Parameters**

I/O	Parameter	Description
I	USB_ER_t *utr	USB transfer structure

(4) **Description**

This function requests data transfer based on the transfer information stored in utr.
When the data transfer ends (the specified data size reached, a short packet received, an error occurred), the callback function is called. The information of the remaining transmit/receive data length, status, and transfer end is set in the argument to this callback function.

(5) **Return value**

Returned Value	Meaning
USB_OK	Success
USB_ERROR	Failure

1 Call this function from the user application or class driver.

2 Set the following members of the structure as arguments: pipe number, start address of transfer data, transfer data length, and the callback function to call atthe end of transfer.

3 If the device is being enumerated, this function returns USB_ERROR until its enumeration is completed.

Example

```
USB_UTR_t  usb_smp_trn_Msg[USB_MAXPIPE_NUM + 1];

USB_ER_t    usb_smp_task(USB_UTR_CB_t complete, uint16_t pipe, uint32_t size, uint8_t *table)
{
        :
        /* Set transfer information */
        trn_msg[pipe].keyword   = pipe;  /* Pipe no. */
        trn_msg[pipe].tranadr   = table; /* Pointer to data buffer */
        trn_msg[pipe].tranlen   = size;  /* Transfer size */
        trn_msg[pipe].complete  = complete;  /* Callback function */

        /* Transfer start request */
        err = R_usb_hstd_TransferStart(&trn_msg[pipe]);

        return err;
        :
}
```

3.14.2 Data transfer forced termination request

R_usb_hstd_TransferEnd

(1) **Synopsis**

Data transfer forced termination request

(2) **C language format**

USB_ER_t R_usb_hstd_TransferEnd(uint16_t pipe_id)

(3) **Parameters**

I/O	Parameter		Description
I	USB_ER_t	pipe_id	Pipe number

(4) **Description**

This function issues a request to forcibly end data transfer via the specified pipes.

(5) **Return value**

Returned Value	Meaning
USB_OK	Success
USB_ERROR	Failure

1 Call this function from the user application or class driver.

Example

```
void  usb_smp_task(void)
{
        uint16_t pipe;
                :
        pipe = USB_PIPEx

        /* Transfer end request */
        err = R_usb_hstd_TransferEnd(pipe);

        return err;
                :
}
```

3.14.3 Host device class driver (HDCD) registration

R_usb_hstd_DriverRegistration

(1) **Synopsis**

Host device class driver (HDCD) registration

(2) **C language format**

void R_usb_hstd_DriverRegistration(USB_HCDREG_t *callback)

(3) **Parameters**

I/O	Parameter	Description
I	USB_HCDREG_t *callback	USB transfer structure

(4) **Description**

This function registers the HDCD information, which is registered in the class driver structure, in the HCD. It then updates the number of registered drivers controlled by the HCD, and registers the HDCD in a new area. The initialization callback function is executed following the completion of registration.

(5) **Return value**

none

1 **Call this function from the user application program or class driver at the initial setup.**

2 **See Table 3-44 USB_HCDREG_t Structure, for details of registration information.**

Example

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

    /* Interface class */
    driver.ifclass      = (uint16_t)USB_IFCLS_VEN;
    /* Target peripheral list */
    driver.tpl          = (uint16_t*)&usb_gapl_devicetpl;
    /* Driver check */
    driver.classcheck   = &usb_hvendor_class_check;
    /* Device configured */
    driver.devconfig    = &usb_hvendor_apl_init;
    /* Device detach */
    driver.devdetach    = &usb_hvendor_apl_close;
    /* Device suspend */
    driver.devsuspend   = &usb_hvendor_apl_suspend;
    /* Device resume */
    driver.devresume    = &usb_hvendor_apl_resume;

    /* Driver registration */
    R_usb_hstd_DriverRegistration(&driver);
}
```

3.14.4 Indicating the completion of class checking

R_usb_hstd_ReturnEnuMGR

(1) **Synopsis**

Indicating the completion of class checking

(2) **C language format**

void R_usb_hstd_ReturnEnuMGR(uint16_t cls_result)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	cls_result	Class check result

(4) **Description**

This function requests continuation of enumeration processing. The MGT analyzes the argument of the function and determines whether or not to initiate a transition of the connected USB device to the configured state.

Specify the result of class checking (USB_OK or USB_ERROR) as the argument (cls_result).

(5) **Return value**

none

1 Call this function at the time of class check processing.

Example

```
void  usb_smp_task(void)
{
      :
      (Enumeration processing)
      :
      R_usb_hstd_ReturnEnuMGR(USB_OK);
}
```

3.14.5 Request for changing the connected device state

R_usb_hstd_ChangeDeviceState

(1) **Synopsis**

Request for changing the connected device state

(2) **C language format**

USB_ER_t R_usb_hstd_ChangeDeviceState(USB_UTR_CB_t complete, int16_t msginfo, uint16_t member)

(3) **Description**

I/O	Parameter		Description
I	USB_UTR_CB_t	complete	Callback function
		msginfo	Message information
		member	Device address/pipe number

(4) **Parameters**

Setting the following values in msginfo and calling this API lead to sending a request to the HCD to change the state of the connected USB device.

Message info	**Outline**
USB_DO_GLOBAL_SUSPEND	Request for transition to the suspended state with remote wakeup enabled (request the device specific suspended state for devices downstream from the hub)
USB_DO_GLOBAL_RESUME	Global resume execution request (resume operation of the connected devices downstream from the ports)
USB_DO_CLR_STALL	STALL clear request

(5) **Return value**

Returned Value	Meaning
USB_OK	Success
USB_ERROR	Failure

1 Call this function from the user application program or class driver

2 When setting USB_DO_CLR_STALL as the argument msginfo, set the pipe number as the argument member

Example

```
void usb_smp_task(void)
{
  R_usb_hstd_ChangeDeviceState(&usb_smp_callback, USB_DO_GROBAL_SUSPEND, devaddr);
}
```

3.14.6 Setting pipe information

R_usb_hstd_SetPipe

(1) **Synopsis**

Setting pipe information

(2) **C language format**

USB_ER_t R_usb_hstd_SetPipe(uint16_t **table)

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t **table	Class check information table pointer（see Table 11-8）

(4) **Description**

This function analyzes the received descriptor and sets pipe information.

(5) **Return value**

Returned Value	Meaning
USB_OK	Success
USB_ERROR	Failure

1 Call this function at the time of class check processing.

2 When the pipe information area is full, this function returns USB_ERROR.

Example

```
void   usb_smp_classcheck(uint16_t **table)
{
       R_usb_hstd_SetPipe(table);
}
```

3.14.7 Acquiring pipe number

R_usb_hstd_GetPipeID

(1) **Synopsis**

Acquiring pipe number

(2) **C language format**

uint16_t R_usb_hstd_GetPipeID(uint16_t devaddr, uint8_t type, uint8_t direction, uint8_t ifnum)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	devaddr	Device address
I	uint8_t	type	Endpoint attribute (USB_EP_ISO / USB_EP_BULK USB_EP_INT)
I	uint8_t	direction	Endpoint direction (USB_EP_IN / USB_EP_OUT)
I	uint8_t	ifnum	Interface number

(4) **Description**

This function returns the pipe number that matches the condition specified by the argument.

(5) **Return value**

Returned Value	Meaning
uint16_t	Pipe number

1 Call this function from the user application program or class driver.

2 Call the function after setting pipe information using R_usb_hstd_SetPipe().

3 When 0xFF is specified as the interface number, the interface number will be ignored when searching.

Example

```
void usb_smp_configured(uint16_t devaddr)
{
        usb_gsmp_pipe = R_usb_hstd_GetPipeID(devaddr, USB_EP_BULK, USB_EP_IN, 0);
}
```

3.14.8 Clearing pipe information

R_usb_hstd_ClearPipe

(1) **Synopsis**

Clearing pipe information

(2) **C language format**

USB_ER_t R_usb_hstd_ClearPipe(uint16_t devaddr)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	devaddr	Device address

(4) **Description**

This function clears the pipe setting of the device address specified by the argument to release the area.

(5) **Return value**

Returned Value	Meaning
USB_OK	Success

1 Call this function from the user application program or class driver.

Example

```
void usb_smp_detach(uint16_t devaddr)
{
        R_usb_hstd_ClearPipe(devaddr);
}
```

3.14.9 Startling MGR task

R_usb_hstd_MgrOpen

(1) **Synopsis**

Starting MGR task

(2) **C language format**

USB_ER_t R_usb_hstd_MgrOpen(void)

(3) **Parameters**

none

(4) **Description**

This function initialize the registration state of MGR.
The value returned is always USB_OK.

(5) **Return value**

I/O	Parameter
USB_OK	Success

1 Call this function from the user application program at the initial setup.

2 Do not call this function after starting the MGR task.

Example

```
void   usb_smp_task(void)
{
       R_usb_hstd_MgrOpen();
}
```

3.14.10 MGR task

R_usb_hstd_MgrTask

(1) **Synopsis**

MGR task

(2) **C language format**

void R_usb_hstd_MgrTask(void)

(3) **Parameters**

none

(4) **Description**

This function provides supplementary functionality between the HCD and HDCD when host controller operation is selected.

(5) **Return value**

1 Call this function in the loop that executes sheduler processing.

Example

```
void   usb_smp_mainloop(void)
{
   while(1)
   {
      /* Start scheduler */
      R_usb_cstd_Scheduler();
      /* Check flag */
      if(USB_FLGSET == R_usb_cstd_CheckSchedule())
      {
          R_usb_hstd_MgrTask();
      }
   }
}
```

3.14.11 HUB class driver(HUBCD) registration

R_usb_hhub_Registration

(1) **Synopsis**

HUB class driver (HUBCD) registration

(2) **C language format**

void R_usb_hhub_Registration (USB_HCDREG_t *callback)

(3) **Parameters**

I/O	Parameter	Description
I	USB_HCDREG_t *callback	Pointer to the USB_HCDREG_t structure

(4) **Description**

This function registers information of the hub class driver (HUBCD)

(5) **Return value**

none

1 Call this function from the user application program at the initial setup.

2 Call this function after setting the the TPL area in the member tpl of the USB_HCDREG_t structure.

Example

```
extern uint16_t  usb_ghhub_TPL[];

void   usb_smp_registration(void)
{
        USB_HCDREG_t  driver;
        :
        driver.tpl = usb_ghhub_TPL;
        R_usb_hhub_Registration(&driver);
        :
}
```

3.14.12 HUB task

R_usb_hhub_Task

(1) **Synopsis**

HUB task

(2) **C language format**

void R_usb_hhub_Task(void)

(3) **Parameters**

none

(4) **Description**

This function manages the states of the downstream ports of a connected USB hub and provides supplementary functionality between the HCD and HDCD.

(5) **Return value**

なし

1 Call this function in the loop that executes scheduler processing.

Example

```
void      usb_smp_mainloop(void)
{
   while(1)
   {
   /* Start scheduler */
   R_usb_cstd_Scheduler();
   /* Check flag */
   if(USB_FLGSET == R_usb_cstd_CheckSchedule())
   {
         :
     R_usb_hhub_Task();
         :
   }
  }
}
```

3.14.13 Sending requests for processing

R_usb_cstd_SndMsg

(1) **Synopsis**

Sending requests for processing

(2) **C language format**

USB_ER_t R_usb_cstd_SndMsg(uint8_t id, USB_MSG_t *mess)

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	id	Task ID to send messages
I	USB_MSG_t	*mess	Message for transmission

(4) **Description**

This function transmits requests for processing to the priority table. This function is defined in the R_USB_SND_MSG macro

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Task ID is not set Task priority is not set Priority table is full (a request cannot be sent to the priority table)

1 Call the scheduler macro (R_USB_SND_MSG) in which this function is registered from the user application program other than the interrupt function or the class driver

2 Specify the address where the memory buffer starts for the message for transmission.

3 The sample program specifies the address where the memory block acquired by the R_PGET_BLK macro starts as that of the message for transmission.

Example

Example

```
void     usb_smp_task()
{
   USB_UTR_t            *p_blf;
   USB_ER_t             err;
     :
   /* Secure a message storage area */
   err = R_USB_PGET_BLK(USB_SMP_MPL, &p_blf);
   if(err == USB_OK)
   {
     /* Set up messages */
     p_blf->msginfo       = USB_SMP_REQ;
     p_blf->keyword       = keyword;
     p_blf->complete      = complete;
   }
   /* Send messages */
   err = R_USB_SND_MSG(USB_SMP_MBX, (USB_MSG_t*)p_blf);
   if(err != USB_OK)
   {
     /* Error processing */
   }
     :
}
```

3.14.14 Checking if the priority table contains a request for processing

R_usb_cstd_RecMsg

(1) **Synopsis**

Checking if the priority table contains a request for processing

(2) **C language format**

USB_ER_t R_usb_cstd_RecMsg(uint8_t id, USB_MSG_t mess)**

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	id	Task ID for reception of messages
I	USB_MSG_t	*mess	Received mesage

(4) **Description**

This function checks if the priority table contains a request for processing corresponding to the task ID. This function is defined in R_USB_RCV_MSG/.

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Task ID is not set

1 Call the scheduler macro in which this function is registered from the user application program or class driver.

2 Do not call this function when the returned value of the R_usb_cstd_CheckSchedule () function is USB_FLGCLR

Example

```
void  usb_smp_task()
{
        USB_UTR_t                   *mess;
        USB_ER_t                    err;
        :
        /* Check the message */
        err = R_USB_RCV_MSG(USB_SMP_MBX, (USB_MSG_t**)&mess);
        if(err == USB_OK)
        {
                /* Processing for the message */
        }
        :
}
```

3.14.15 Securing an area for storing requests for processing

R_usb_cstd_PgetBlk

(1) **Synopsis**

Securing an area for storing requests for processing

(2) **C language format**

USB_ER_t R_usb_cstd_PgetBlk(uint8_t id, USB_UTR_t blk)**

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	id	Task ID for securing an area
I	USB_MSG_t	**blk	Pointer to the area to be secured

(4) **Description**

This function secures an area for storing messages. This API secures 40 bytes per block. This function is registered in R_USB_PGET_BLK.

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Task ID is not set The area is not secured

1 Call the scheduler macro in which this function is registered from the user application program or class driver.

Example

```
void     usb_smp_task()
{
   USB_UTR_t            *p_blf;
   USB_ER_t             err;
     :
   /*Secure a message storage area */
   err = R_USB_PGET_BLK(USB_SMP_MPL, &p_blf);
   if(err == USB_OK)
   {
     /* Set up messages */
     p_blf->msginfo       = USB_SMP_REQ;
     p_blf->keyword       = keyword;
     p_blf->complete      = complete;
   }
   /* Send messages */
   err = R_USB_SND_MSG(USB_SMP_MBX, (USB_MSG_t*)p_blf);
   if(err != USB_OK)
   {
     /* Error processing */
   }
     :
}
```

3.14.16 Releasing an area for storing requests for processing

R_usb_cstd_RelBlk

(1) **Synopsis**

Releasing an area for storing requests for processing

(2) **C language format**

USB_ER_t R_usb_cstd_RelBlk(uint8_t id, USB_UTR_t* blk)

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	id	Task ID for releasing an area
I	USB_MSG_t	*blk	Pointer to the area to be released

(4) **Description**

This function releases an area for storing messages. The function is registered in R_USB_REL_BLK

(5) **Return value**

Returned Value	Meaning
USB_OK	The area is released
USB_ERROR	Task ID is not set The area is not released

1 Call the scheduler macro in which this function is registered from the user application program or class driver.

Example

```
void  usb_smp_task()
{
      USB_ER_t                  err;
      USB_UTR_t                 *mess;
      :
      /* Release the message storage area */
      err = R_USB_REL_BLK(USB_SMP_MPL, mess);
      if(err != USB_OK)
      {
            /* Error processing */
      }
      :
}
```

3.14.17 Managing requests for processing

R_usb_cstd_Scheduler

(1) **Synopsis**

Managing requests for processing

(2) **C language format**

void R_usb_cstd_Scheduler(void)

(3) **Parameters**

none

(4) **Description**

This function manages requests for processing from the individual tasks

(5) **Return value**

none

1 Call this function in the loop that executes scheduler processing for non-OS operations.

Example

```
void    usb_smp_mainloop(void)
{
   while(1)
   {
   /* Check flag */
   if(USB_FLGSET == R_usb_cstd_CheckSchedule())
   {
         :
     /* Task processing */
         :
   }
   /* Start scheduler */
   R_usb_cstd_Scheduler();
         :
}
```

3.14.18 Setting the priority of tasks

R_usb_cstd_SetTaskPri

(1) **Synopsis**

Setting the priority of tasks

(2) **C language format**

void R_usb_cstd_SetTaskPri(uint8_t tasknum, uint8_t pri)

(3) **Parameters**

I/O	Parameter		Description
I	uint8_t	tasknum,	Task ID
I	uint8_t	pri	Task priority

(4) **Description**

This function sets the priority of tasks. The priority (pri) specified by the 2nd argument is set in the task specified by the 1st argument (tasknum).

(5) **Return value**

none

1 Call this function from the user application program at the initial setup. Messages can be transferred by setting the task ID priorities.

Example

```
void usb_smp_driver_start(void)
{
   /* Set the priority of the sample task to 4 */
   R_usb_cstd_SetTaskPri(USB_SMP_TSK, USB_PRI_4);
}
```

3.14.19 Checking whether or not processing is scheduled

R_usb_cstd_CheckSchedule

(1) **Synopsis**

Checking whether or not processing is scheduled

(2) **C language format**

uint8_t R_usb_cstd_CheckSchedule(void)

(3) **Parameters**

none

(4) **Description**

This function checks if the registered task contains a received message

(5) **Return value**

Returned Value	Meaning
USB_FLGSET	Request for processing
USB_FLGCLR	No request for processing

1 Call this function in the loop that executes scheduler processing for non-OS operations.

Example

```
void    usb_smp_mainloop(void)
{
   while(1)
   {
   /* Check flag */
   if(USB_FLGSET == R_usb_cstd_CheckSchedule())
   {
         :
     /* Task processing */
         :
   }
   /* Start scheduler */
   R_usb_cstd_Scheduler();
         :
}
```

3.14.20 Host Mass Storage Class task

R_usb_hmsc_Task

(1) **Synopsis**

Host Mass Storage Class task

(2) **C language format**

void R_usb_hmsc_Task(void)

(3) **Parameters**

none

(4) **Description**

This function is a task for HMSCD.

This function controls BOT

(5) **Return value**

none

1 Please call this function from a loop that executes the scheduler processing.

Example

```
void    usb_smp_mainloop(void)
{
   while(1)
   {
   /* Scheduler processing */
   R_usb_cstd_Scheduler();
   /* Checking flag */
   if(USB_FLGSET == R_usb_cstd_CheckSchedule())
   {
    R_usb_hstd_MgrTask();
    R_usb_hhub_Task();
   R_usb_hmsc_Task();
   }
}
```

3.14.21 HMSC driver start

R_usb_hmsc_driver_start

(1) **Synopsis**

HMSC driver start

(2) **C language format**

void R_usb_hmsc_driver_start(void)

(3) **Parameters**

none

(4) **Description**

This function sets the priority of HMSC driver task.

(5) **Return value**

none

1 Call this function from the user application program during initialization

Example

```
void usb_hstd_task_start( void )
{
   R_usb_hmsc_driver_start();       /* Host Class Driver Task Start Setting */
}
```

3.14.22 Check descriptor

R_usb_hmsc_ClassCheck

(1) **Synopsis**

Check descriptor

(2) **C language format**

void R_usb_hmsc_ClassCheck(uint16_t **table)

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t **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

(4) **Description**

This is a class driver registration function. It is registered to the driver registration structure member classcheck, as a callback function during HMSC 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.

(5) **Return value**

none

Example

```
void    usb_hapl_registration(USB_UTR_t *ptr)
{
    USB_HCDREG_t driver;

   /* Driver check */
   driver.classcheck = &R_usb_hmsc_ClassCheck;
}
```

3.14.23 Returns HMSCD operation state

R_usb_hmsc_GetDevSts

(1) **Synopsis**

Returns HMSCD operation state

(2) **C language format**

uint16_t R_usb_hmsc_GetDevSts(uint16_t side)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	side	Drive number

(4) **Description**

Returns HMSCD operation state

(5) **Return value**

Returned Value		Meaning
usb_ghmsc_AttSts	SB_HMSC_DEV_ATT	CONNECT
usb_ghmsc_AttSts	SB_HMSC_DEV_DET	DisCONNECT

1 The argument side, specify the drive number assigned by R_usb_hmsc_alloc_drvno()。

Example

```
void    usb_smp_task(void)
{
  /* Checking the device state */
  if(R_usb_hmsc_GetDevSts(drvno) == USB_HMSC_DEV_DET)
  {
    /* Detach processing */
  }
}
```

3.14.24 Issue READ10 command

R_usb_hmsc_Read10

(1) **Synopsis**

Issue READ10 command

(2) **C language format**

uint16_t R_usb_hmsc_Read10(uint16_t side, uint8_t *buff, uint32_t secno,
uint16_t seccnt, uint32_t trans_byte)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	side	Drive number
I	uint8_t	*buff	Read data area
I	uint32_t	secno	Sector number
I	uint16_t	seccnt	Sector count
I	uint32_t	trans_byte	Transfer data length

(4) **Description**

Creates and executes the READ10 command.

When a command error occurs, the REQUEST_SENSE command is executed to get error information.

(5) **Return value**

Returned Value	Meaning
-	Error code

Example

```
void      usb_smp_task(void)
{
   uint16_t   result;

   /* Issuing READ10 */
   result = R_usb_hmsc_Read10(side, buff, secno, seccnt, trans_byte);
   if(result != USB_HMSC_OK)
   {
      /* Error Processing */
   }
}
```

3.14.25 Issue WRITE10 command

R_usb_hmsc_Write10

(1) **Synopsis**

Issue WRITE10 command

(2) **C language format**

uint16_t R_usb_hmsc_Write10(uint16_t side, const uint8_t *buff, uint32_t secno, uint16_t seccnt, uint32_t trans_byte)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	side	Drive number
I	uint8_t	*buff	Write data area
I	uint32_t	secno	Sector number
I	uint16_t	seccnt	Sector count
I	uint32_t	trans_byte	Transfer data length

(4) **Description**

Creates and executes the WRITE10 command.

When a command error occurs, the REQUEST_SENSE command is executed to get error information.

(5) **Return value**

Returned Value	Meaning
-	Error code

Example

```
void    usb_smp_task(void)
{
  uint16_t  result;

  /* Issuing WRITE10 */
  result = R_usb_hmsc_Write10(side, buff, secno, seccnt, trans_byte);
  if(result != USB_HMSC_OK)
  {
    /* Error processing */
  }
}
```

3.14.26 Issue GetMaxLUN request

R_usb_hmsc_GetMaxUnit

(1) **Synopsis**

Issue GetMaxLUN request

(2) **C language format**

USB_ER_t R_usb_hmsc_GetMaxUnit(uint16_t addr, USB_CB_t complete)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	addr	Device address
I	USB_CB_t	complete	Callback function

(4) **Description**

This function issues the GET_MAX_LUN request and gets the maximum storage unit count. The callback function which is specified in the argument (complete) is called when completing this request

(5) **Return value**

Returned Value	Meaning
USB_OK	GET_MAX_LUN issued
USB_ERROR	GET_MAX_LUN not issued

Example

```
void    usb_smp_task(void)
{
  USB_ER_t    err;

  /* Getting Max unit number */
  err = R_usb_hmsc_GetMaxUnit(devadr, usb_hmsc_StrgCheckResult);
  if(err == USB_ERROR)
  {
    /* Error processing */
  }
}
```

3.14.27 Issue Mass Storage Reset request.

R_usb_hmsc_MassStorageReset

(1) **Synopsis**

Issue Mass Storage Reset request.

(2) **C language format**

USB_ER_t R_usb_hmsc_MassStorageReset(uint16_t addr, USB_CB_t complete)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	addr	Device address
I	USB_CB_t	complete	Callback function

(4) **Description**

This function issues the MASS_STORAGE_RESET request and cancels the protocol error. The callback function which is specified in the argument (complete) is called when completing this request.

(5) **Return value**

Returned Value	Meaning
USB_OK	MASS_STORAGE_RESET issued
USB_ERROR	MASS_STORAGE_RESET not issued

Example

```
void    usb_smp_task(void)
{
  USB_ER_t    err;

  /* Cansel the protocol error */
  err = R_usb_hmsc_MassStorageReset(devadr, usb_hmsc_CheckResult);
  if(err == USB_ERROR)
  {
    /* Error processing */
  }
}
```

3.14.28 Allocates the drive number

R_usb_hmsc_alloc_drvno

(1) **Synopsis**

Allocates the drive number

(2) **C language format**

uint16_t R_usb_hmsc_alloc_drvno(uint16_t *side, uint16_t devadr)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	*side	Drive number pointer
I	uint16_t	devadr	Device address of MSC device

(4) **Description**

This function allocate the drive number to the connected MSC device, and store in the argument (*side).

Drive number is assigned from 0 in the order

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

Example

```
void    usb_smp_task(void)
{
    /* Allocates the drive number */
    R_usb_hmsc_alloc_drvno(&drvno, devadr);
}
```

3.14.29 Frees the drive number

R_usb_hmsc_free_drvno

(1) **Synopsis**

Frees the drive number

(2) **C language format**

uint16_t R_usb_hmsc_free_drvno(uint16_t side)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	side	Drive number

(4) **Description**

This function frees the drive number which is specified by the argument.

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

Example

```
void    usb_smp_task(void)
{

    /* Frees the drive number */
    R_usb_hmsc_free_drvno(drvno);
}
```

3.14.30 Refers the drive number

R_usb_hmsc_ref_drvno

(1) **Synopsis**

Refers the drive number

(2) **C language format**

uint16_t R_usb_hmsc_ref_drvno(uint16_t *side, uint16_t devadr)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	*side	Drive No Pointer
I	uint16_t	devadr	Device address

(4) **Description**

This function refers the drive numer based on the device address which are specified by the argument (devadr), and stores in the argument (*side).

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

Example

```
void    usb_smp_task(void)
{
    /* refers the drive number */
    R_usb_hmsc_ref_drvno(&drvno, devadr);
}
```

3.14.31 Storage drive task

R_usb_hmsc_StrgDriveTask

(1) **Synopsis**

Storage drive task

(2) **C language format**

void R_usb_hmsc_StrgDriveTask(void)

(3) **Parameters**

none

(4) **Description**

This API does the processing to get the storage device information by sending storage command to the storage device.

(5) **Return value**

none

1 Please call this function from a loop that executes the scheduler processing

Example

```
void usb_hapl_mainloop(void)
{
   while(1)
   {
      R_usb_cstd_Scheduler();

      if(USB_FLGSET == R_usb_cstd_CheckSchedule())
      {
            R_usb_hstd_MgrTask();
            R_usb_hhub_Task();
            R_usb_hmsc_Task();
            R_usb_hmsc_StrgDriveTask();
      }
   }
}
```

3.14.32 Search drive

R_usb_hmsc_StrgDriveSearch

(1) Synopsis

Search drive

(2) C language format

uint16_t R_usb_hmsc_StrgDriveSearch(uint16_t addr, USB_UTR_CB_t complete)

(3) Parameters

I/O	Parameter		Description
I	uint16_t	addr	USB address
I	USB_UTR_CB_t	complete	Callback function

(4) Description

This API checks the follows by sending command to USB device which is specified in the argument (addr).

The callback function which is specified in the argument (complete) is called whencompleting the drive searching.

Refer to「11.9.4 Acquiring Information on the USB Storage Devices」

(5) Return value

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

1 Continue to operate the "Scheduler" from call this API until the callback function is called and do not the other USB processing.

Example

```
/* Callback function */
void usb_hmsc_StrgCommandResult( USB_UTR_t *mess )
{
      :
}

void usb_hmsc_SampleAplTask(void)
{
   R_usb_hmsc_StrgDriveSearch(addr, &usb_hmsc_StrgCommandResult);
}
```

3.14.33 Open drive

R_usb_hmsc_StrgDriveOpen

(1) **Synopsis**

Open drive

(2) **C language format**

uint16_t R_usb_hmsc_StrgDriveOpen(uint16_t *side, uint16_t addr)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	*side	Description
I	uint16_t	addr	Drive No Pointer

(4) **Description**

Open the address specified in the argument.

Call after the enumeration is complete.

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

1 Use the R_usb_hmsc_alloc_drvno () in this function inside to assign the drive number.

2 Use the R_usb_hstd_GetPipeID() in this function inside to set the pipe number.

Example

```
void msc_configured(uint16_t devadr)
{
   R_usb_hmsc_StrgDriveOpen(&drvno, devadr);
}
```

3.14.34 Close drive

R_usb_hmsc_StrgDriveClose

(1) **Synopsis**

Close drive

(2) **C language format**

uint16_t R_usb_hmsc_StrgDriveClose(uint16_t side)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	side	Drive Number

(4) **Description**

Close the drive specified by the argument.

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

1 Use the R_usb_hmsc_free_drvno() in this function inside to open the drive number.

2 Use the R_usb_hstd_ClearPipe() in this function inside to clear the pipe information.

Example

```
void msc_detach(uint16_t devadr)
{
   R_usb_hmsc_StrgDriveClose(drv_no);
}
```

3.14.35 Read Sector

R_usb_hmsc_StrgReadSector

(1) **Synopsis**

Read Sector

(2) **C language format**

uint16_t R_usb_hmsc_StrgReadSector(uint16_t side, uint8_t *buff
, uint32_t secno, uint16_t seccnt, uint32_t trans_byte)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	side	Drive number
I	uint8_t	*buff	Pointer to read data storage area
I	uint32_t	secno	Sector number
I	uint16_t	seccnt	Sector count
I	uint32_t	trans_byte	Transfer data length

(4) **Description**

Reads the sector information of the drive specified by the argument.

An error response occurs in the following cases.

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

1 Please call this function from FAT library I/F function.

2 Use the R_usb_hmsc_Read10 () in this function inside to read sector information.

3 Use the R_usb_hmsc_GetDevSts () in this function inside, it has confirmed the connection status.If in a disconnected state, and terminates with an error before writing

Example

```
DRESULT disk_read(BYTE pdrv, BYTE* buff, DWORD sector, UINT count)
{
    uint32_t       err;
    uint32_t       tran_byte;

    /* set transfer length */
    tran_byte = count * _MIN_SS;

    /* read function */
    err = R_usb_hmsc_StrgReadSector(pdrv, buff, sector, (uint16_t)count, tran_byte);
    if (err != USB_OK)
    {
        return RES_ERROR;
    }
}
```

3.14.36 Write Sector Information

R_usb_hmsc_StrgWriteSector

(1) **Synopsis**

Write Sector Information

(2) **C language format**

uint16_t R_usb_hmsc_StrgWriteSector(uint16_t side, const uint8_t *buff , uint32_t secno, uint16_t seccnt, uint32_t trans_byte)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	side	Drive number
I	uint8_t	*buff	Pointer to read data storage area
I	uint32_t	secno	Sector number
I	uint16_t	seccnt	Sector count
I	uint32_t	trans_byte	Transfer data length

(4) **Description**

Writes the sector information of the drive specified by the argument.

An error response occurs in the following cases.

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

1 Please call this function from FAT library I/F function.

2 Use the R_usb_hmsc_Read10 () in this function inside to read sector information.

3 Use the R_usb_hmsc_GetDevSts () in this function inside, it has confirmed the connection status.If in a disconnected state, and terminates with an error before writing.

Example

```
DRESULT disk_read(BYTE pdrv, BYTE* buff, DWORD sector, UINT count)
{
   uint32_t       err;
   uint32_t       tran_byte;

   /* set transfer length */
   tran_byte = count * _MIN_SS;

   /* read function */
   err = R_usb_hmsc_StrgReadSector(pdrv, buff, sector, (uint16_t)count, tran_byte);
   if (err != USB_OK)
   {
      return RES_ERROR;
   }
}
```

3.14.37 Check Read/Write end

R_usb_hmsc_StrgCheckEnd

(1) Synopsis

Check Read/Write end

(2) **C language format**

uint16_t R_usb_hmsc_StrgCheckEnd(void)

(3) **Parameters**

none

(4) **Description**

Return status of read / write sequence.

(5) **Return value**

Returned Value	Meaning
USB_FALSE	Reading / writing
USB_TRUE	Read / write complete
USB_ERROR	Read / write error

1 Call this API function periodically after calling R_usb_hmsc_StrgReadSector() or R_usb_hmsc_StrgWriteSector().

Example

```
DRESULT disk_write(BYTE pdrv, const BYTE* buff, DWORD sector, UINT count)
{
    uint32_t        err;

    /* write function */
    R_usb_hmsc_StrgWriteSector(pdrv, buff, sector, (uint16_t)count, tran_byte);

    /* Wait USB write sequence(WRITE10) */
    do
    {
        R_usb_cstd_Scheduler();
        if (R_usb_cstd_CheckSchedule() == USB_FLGSET)
        {
            R_usb_hstd_MgrTask();       /* MGR task */
            R_usb_hhub_Task();          /* HUB task */
            R_usb_hmsc_task();          /* HMSC Task */
            R_usb_hmsc_StrgDriveTask(); /* HSTRG Task */
        }
        err = R_usb_hmsc_StrgCheckEnd();
    }
    while (err == USB_FALSE);

    /* Set transfer result */
    if (err != USB_TRUE)
    {
        return RES_ERROR;
    }
    else
    {
        return RES_OK;
    }
}
```

3.14.38 Issue Storage Command

R_usb_hmsc_StrgUserCommand

(1) **Synopsis**

Issue Storage Command

(2) **C language format**

uint16_t R_usb_hmsc_StrgUserCommand(uint16_t side, uint16_t command , uint8_t *buff, USB_UTR_CB_t complete)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	side	Drive number
I	uint16_t	command	Command to be issued
I	uint8_t	*buff	Data pointer
I	USB_CB_t	complete	Callback function

(4) **Description**

This function issues the storage command specified by the given argument, to HMSC driver. The callback function which is specified in the argument (complete) is called when completing the issued storage command. Here are the storage commands supported:

Storage commands	Description
USB_ATAPI_TEST_UNIT_READY	Check status of peripheral device
USB_ATAPI_REQUEST_SENCE	Get status of peripheral device
USB_ATAPI_INQUIRY	Get parameter information of logical unit
USB_ATAPI_MODE_SELECT6	Specify parameters
USB_ATAPI_PREVENT_ALLOW	Enable/disable media removal
USB_ATAPI_READ_FORMAT_CAPACITY	Get formattable capacity
USB_ATAPI_READ_CAPACITY	Get capacity information of logical unit
USB_ATAPI_MODE_SENSE10	Get parameters of logical unit

(5) **Return value**

Returned Value	Meaning
USB_OK	Normal end
USB_ERROR	Error end

1 Use the API function of HMSCD in this function inside to isuue the storage command.

2 Use the R_usb_hmsc_GetDevSts () in this function inside, it has confirmed the connection status.If in a disconnected state, and terminates with an error before writing.

Example

```
/* Callback function */
void  strgcommand_complete(USB_UTR_t *mess)
{
   :
}
void   usb_smp_task(void)
{
   :
   /* Issuing TEST_UNIT_READY */
   err = R_usb_hmsc_StrgUserCommand(side, USB_ATAPI_TEST_UNIT_READY, buf,
   strgcommand_complete);
   :
}
```

3.14.39 Get device status

disk_status

(1) **Synopsis**

Get device status

(2) **C language format**

DSTATUS disk_status(BYTE pdrv)

(3) **Parameters**

I/O	Parameter		Description
I	BYTE	pdrv	Physical drive number (0-9)

(4) **Description**

This function return the value of the *R_usb_disk_status [pdrv]*.

If more than 10 is set to pdrv, return (*STA_NOINIT* | *STA_NODISK)*.

(5) **Return value**

The current drive status is returned in combination of status flags described below

Returned Value	Meaning
STA_NOINIT	Indicates that the device is not initialized
STA_NODISK	Indicates that no medium in the drive.
STA_PROTECT	Indicates that the medium is write protected. (not use in sample)

3.14.40 Initialize device

disk_initialize

(1) **Synopsis**

Initialize device

(2) **C language format**

DSTATUS disk_initialize(BYTE pdrv)

(3) **Parameters**

I/O	Parameter		Description
I	BYTE	pdrv	Physical drive number (0-9)

(4) **Description**

This function return the value of the *R_usb_disk_status [pdrv]*.

(5) **Return value**

The current drive status is returned in combination of status flags described below.

Returned Value	Meaning
STA_NOINIT	Indicates that the device is not initialized
STA_NODISK	Indicates that no medium in the drive.
STA_PROTECT	Indicates that the medium is write protected. (not use in sample)

1 This function is under control of FatFs module. Application program MUST NOT call this function, or FAT structure on the volume can be broken.

2 To re-initialize the file system, use f_mount function instead.

3.14.41 Disk read

disk_read

(1) **Synopsis**

Read Sector(s)

(2) **C language format**

DRESULT disk_read(BYTE pdrv, BYTE* buff, DWORD sector, UINT count)

(3) **Parameters**

I/O	Parameter		Description
I	BYTE	pdrv	Physical drive number
O	BYTE	*buff	Pointer to the read data buffer
I	DWORD	sector	Start sector number
I	UINT	count	Number of sectros to read

(4) **Description**

This function call the API function *R_usb_hmsc_StrgReadSector()* of HMSDD.

Arguments setting of *R_usb_hmsc_StrgReadSector()* is as follows.

Argument	value
uint16_t side	pdrv
uint8_t *buff	buff
uint32_t secno	sector
uint16_t seccnt	(uint16_t)count
uint32_t trans_byte	count * _MIN_SS

This function works the scheduler loops in this function until the USB read sequence is completed.

If it detects a USB disconnect in the middle, and then return the RES_ERROR.

(5) **Return value**

Returned Value	Meaning
RES_OK	The function succeeded.
RES_ERROR	Any hard error occured during the read operation
RES_PARERR	The device has not been initialized. (not use in sample)
RES_NOTRDY	The function succeeded.

3.14.42 Disk write

disk_write

(1) **Synopsis**

Write Sector(s)

(2) **C language format**

DRESULT disk_write(BYTE pdrv, const BYTE* buff, DWORD sector, UINT count)

(3) **Parameters**

I/O	Parameter		Description
I	BYTE	pdrv	Physical drive number
I	BYTE	*buff	Pointer to the data to be written
I	DWORD	sector	Start sector number
I	UINT	count	Number of sectros to write

(4) **Description**

This function call the API function *R_usb_hmsc_StrgWriteSector()* of HMSDD.

Arguments setting of *R_usb_hmsc_StrgWriteSector()* is as follows.

Argument	value
uint16_t side	pdrv
const uint8_t *buff	buff
uint32_t secno	sector
uint16_t seccnt	(uint16_t)count
uint32_t trans_byte	count * _MIN_SS

This function works the scheduler loops in this function until the USB write sequence is completed.

If it detects a USB disconnect in the middle, and then return the RES_ERROR.

(5) **Return value**

Returned Value	Meaning
RES_OK	The function succeeded.
RES_ERROR	Any hard error occured during the write operation.
RES_PARERR	Invalid parameter. (not use in sample)
RES_NOTRDY	The device has not been initialized. (not use in sample)

3.14.43 Disk ioctl

disk_ioctl

(1) Synopsis

Control device dependent features

(2) C language format

DRESULT disk_ioctl(BYTE pdrv, BYTE cmd, void* buff)

(3) Parameters

I/O	Parameter		Description
I	BYTE	pdrv	Physical drive number
I	BYTE	cmd	Control command code
I/O	uint8_t	*buff	Parameter and data buffer

(4) Description

This function return RES_OK without the processing for all of the command.

(5) Return value

Returned Value	Meaning
RES_OK	The function succeeded.
RES_ERROR	An error occured. (not use in sample)
RES_PARERR	The command code or parameter is invalid. (not use in sample)
RES_NOTRDY	The device has not been initialized. (not use in sample)

3.14.44 Get current time

get_fattime

(1) **Synopsis**

Get current time

(2) **C language format**

DWORD get_fattime(void)

(3) **Parameters**

none

(4) **Description**

This function return 0x00000000 without setting the date and time information.

(5) **Return value**

Currnet local time is returned with packed into a DWORD value. The bit field is as follows:

Returned Value	Meaning
bit31:25	Year origin from the 1980 (0..127)
bit24:21	Month (1..12)
bit20:16	Day of the month(1..31)
bit15:11	Hour (0..23)
bit10:5	Minute (0..59)
bit4:0	Second / 2 (0..29)

4. CAN Sample Software

4.1 Overview

This section describes the sample program that performs communication by using the CAN driver that controls the CAN controller 1 channel (CAN1) installed on the evaluation board.

The features of the CAN interface sample program are as follows.

CAN communication can be tested by connecting to a personal computer and by using terminal software.

- Transmission functions
  - Message transmission by using a transmission buffer
  - Message transmission by using a transmission/reception FIFO buffer (transmission mode)
- Reception functions
  - Message reception by using a reception buffer
  - Message reception by using a reception FIFO buffer
  - Message reception by using a transmission/reception FIFO buffer (reception mode)
- Test functions
  - Self-test mode 0 (external loopback mode)
    - Transmission buffer to reception buffer
    - Transmission buffer to reception FIFO buffer
    - Transmission buffer to transmission/reception FIFO buffer (reception mode)
    - Transmission/reception FIFO buffer (transmission mode) to transmission/reception FIFO buffer (reception mode)
  - Self-test mode 1 (internal loopback mode)
    - Transmission buffer to reception buffer
    - Transmission buffer to reception FIFO buffer
    - Transmission buffer to transmission/reception FIFO buffer (reception mode)
    - Transmission/reception FIFO buffer (transmission mode) to transmission/reception FIFO buffer (reception mode)
- Transfer rate
  - 1 Mbps, 500 Kbps and125 Kbps are supported (selectable from the menu).

Restrictions

The following restrictions apply to this sample program:

(1) The used channel is fixed to CAN1.

(2) The message format is fixed. (Data frames with a standard ID (0x120))

(3) The reception rule is fixed.

Reception rule table page number: 0

Number of reception rules: 1 (table number: 0)

Reception rule ID: Data frames with a standard ID (0x120)

(4) The used buffers are fixed.

Transmission buffer number: 0

Reception buffer number: 1

Transmission/reception FIFO buffer number: 0

Transmission/reception FIFO buffer number (transmission mode): 0

Transmission buffer number to be linked to the transmission/reception FIFO buffer (transmission mode): 2

Transmission/reception FIFO buffer number (reception mode): 1

(5) Other information

The following features are not supported:

Transmission abort, transmission by transmission queue, transmission history function, gateway function, test functions (standard test mode, listen-only mode, RAM test, inter-channel communication test), and error detection/correction of RSCAN RAM

Target Devices

EC-1

When applying the sample program covered in this application note to another microcontroller, modify the program according to the specifications for the target microcontroller and conduct an extensive evaluation of the modified program.

4.2 Structures, Unions and Enumerated Types

The following shows the structures, unions, and enumerated types used in this sample code.

Table 4-1 tx_data_t Structure

Member Name	Description
uint32_t num;	Number
uint8_t * data;	Data pointer
uint32_t send_num;	Number of send

Table 4-2 rx_data_t Structure

Member Name	Description
uint32_t num;	Number
uint8_t * data;	Data pointer

4.3 Functions

A list of functions is shown in Table 4-3 List of Functions.

Table 4-3 List of Functions

Function Name	Description	Scope	Definition File
main	Main processing	global	main.c
can_main_init	CAN sample initialization process	local	main.c
ecm_init	ECM setting initialization	local	main.c
icu_init	Interrupt setting	local	main.c
port_init	Initializes the port settings	local	main.c
soft_wait	Software wait	local	main.c
scifa_interrupt_source	Enables SCIFA interrupt processing	local	main.c
key_handler_callback	Callback function (SCIFA reception-FIFO-data-full interrupt)	local	main.c
menu	CAN test menu	local	main.c
self_menu	CAN self-test menu	local	main.c
test_end	CAN test end	local	main.c
can1_open	Opens the CAN module	local	main.c
user_callback_entry	Registers a callback function information structure	local	main.c
select_interrupt_source	Registers interrupts and callback functions	local	main.c
interrupt_disable	Disables interrupt processing	local	main.c
creat_message_header	Creates sample message headers	local	main.c
user_gl_err_callback	Callback function (CAN global error)	local	main.c
user_ch1_err_callback	Callback function (CAN1 error)	local	main.c
user_rx_fifo_callback	Callback function (CAN reception FIFO interrupt)	local	main.c
user_ch1_tx_callback	Callback function (CAN1 transmission interrupt)	local	main.c
user_ch1_rx_fifo_callback	Callback function (CAN1 transmission/reception FIFO reception completion interrupt)	local	main.c
tx_demo_buffer	Message transmission by using a transmission buffer	local	main.c
tx_demo_fifo	Message transmission by using a transmission/reception FIFO buffer (transmission mode)	local	main.c
rx_demo_buffer	Message reception by using a reception buffer	local	main.c
rx_demo_fifo	Message reception by using a transmission/reception FIFO buffer (reception mode)	local	main.c
rx_demo_rx_fifo	Message reception by using a reception FIFO buffer	local	main.c
trx_demo_fifo	Message transmission test while receiving a message	local	main.c
selftest_buf_to_buf	Test which sends a message by using the transmission buffer and receives the message by the reception buffer	local	main.c
selftest_buf_to_rx_fifo	Test which sends a message by using the transmission buffer and receives the message by the reception FIFO buffer	local	main.c
selftest_buf_to_fifo	Test which sends a message by using the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode)	local	main.c

Function Name	Description	Scope	Definition File
selftest_fifo_to_fifo	Test which sends a message by using the transmission/reception FIFO buffer (transmission mode) and receives the message by the transmission/reception FIFO buffer (reception mode)	local	main.c
demo_result	Outputs the result of processing	local	main.c
demo_init	Initializes the variables and others for use in this API function	local	main.c
demo_end	Clears the information about the buffer used	local	main.c
rx_rule	Sets CAN reception rules	local	main.c
send_termination_code	Sends a message end code	local	main.c
output_tx_msg_format	Outputs the format of the message for transmission	local	main.c
output_tx_msg_data	Outputs the message for transmission	local	main.c
output_rx_msg_data	Outputs the received message	local	main.c
output_rx_msg_format	Outputs the format of the received messages	local	main.c
clear_status	Clears the flags used in the sample program	local	main.c
write_buffer	Writes messages for transmission to the register related to the transmission buffer	local	main.c
write_fifo	Writes messages for transmission to the register related to the transmission/reception FIFO buffer	local	main.c
read_buffer	Reads received messages from the reception buffer	local	main.c
read_rx_fifo	Reads received messages from the reception FIFO buffer	local	main.c
read_fifo	Reads received messages from the transmission/reception FIFO buffer	local	main.c
set_fifo_buffer	Registers the transmission/reception FIFO buffer	local	main.c
rx_fifo_mode	Registers the reception FIFO buffer	local	main.c
get_error_status	Gets the interrupt source	local	main.c

4.4 Details of the Functions

4.4.1 main

(1) **Synopsis**

Main processing of the sample program

(2) **C language format**

int main (void);

(3) **Parameters**

None

(4) **Description**

This function is the main processing of sample program.

For details on the processing, see section 4.5.1, Main Processing.

(5) **Returned values**

Returned Value	Meaning
0	End of CAN sample program

4.4.2 can_main_init

(1) **Synopsis**

CAN sample initialization process.

(2) **C language format**

static void can_main_init (void);

(3) **Parameters**

None

(4) **Description**

Initialize the variable

(5) **Returned values**

None

4.4.3 ecm_init

(1) **Synopsis**

Initialize ECM settings.

(2) **C language format**

static void ecm_init (void);

(3) **Parameters**

None

(4) **Description**

Initialize ECM settings.

(5) **Returned values**

None

4.4.4 icu_init

(1) **Synopsis**

Interrupt setting.

(2) **C language format**

static void icu_init (void);

(3) **Parameters**

None

(4) **Description**

Enable interrupt handling.

(5) **Returned values**

None

4.4.5 port_init

(1) **Synopsis**

Initializes the port settings.

(2) **C language format**

static void port_init(void);

(3) **Parameters**

None

(4) **Description**

This function initializes the port settings.

(5) **Returned values**

None

4.4.6 soft_wait

(1) **Synopsis**

Software wait processing

(2) **C language format**

static void soft_wait(void);

(3) **Parameters**

None

(4) **Description**

This function performs software wait processing by using the NOP command.

(5) **Returned values**

None

4.4.7 scifa_interrupt_source

(1) **Synopsis**

Enables SCIFA interrupt processing.

(2) **C language format**

static void scifa_interrupt_source(void);

(3) **Parameters**

None

(4) **Description**

This function enables SCIFA interrupt processing (reception FIFO data full (RDF)).

(5) **Returned values**

None

4.4.8 key_handler_callback

(1) **Synopsis**

Callback function (SCIFA receive-FIFO-data-full interrupt)

(2) **C language format**

static void key_handler_callback(void);

(3) **Parameters**

None

(4) **Description**

This is a callback function executed when SCIFA receive-FIFO-data-full interrupt is generated.

(5) **Returned values**

None

4.4.9 menu

(1) **Synopsis**

CAN test menu

(2) **C language format**

static void menu(void);

(3) **Parameters**

None

(4) **Description**

This function displays the CAN test menu.

For the displayed menu, see section 4.6.5, Sample Program Functions.

(5) **Returned values**

None

4.4.10 self_menu

(1) **Synopsis**

CAN self-test menu

(2) **C language format**

static void self_menu(void);

(3) **Parameters**

None

(4) **Description**

This function displays the self-test menu when [7] is selected in the main menu.
For the displayed menu, see section 4.6.5, Sample Program Functions.

(5) **Returned values**

None

4.4.11 test_end

(1) **Synopsis**

CAN test end

(2) **C language format**

static void test_end(void);

(3) **Parameters**

None

(4) **Description**

This function closes the CAN module and displays an end message.

(5) **Returned values**

None

4.4.12 can1_open

(1) **Synopsis**

Opens the CAN module.

(2) **C language format**

static void can1_open(void);

(3) **Parameters**

None

(4) **Description**

This function opens the CAN module.

(5) **Returned values**

None

4.4.13 user_callback_entry

(1) Synopsis

Registers a callback function information structure.

(2) C language format

static void user_callback_entry (void);

(3) Parameters

None

(4) Description

This function registers the name of the callback function used as a member of the can_callback_t structure (callback function information structure).

Write null if this function is not used.

(5) Returned values

None

4.4.14 select_interrupt_source

(1) **Synopsis**

Registers the callback function.

(2) **C language format**

static void select_interrupt_source(void);

(3) **Parameters**

None

(4) **Description**

This function registers the callback function specified in user_callback_entry().

(5) **Returned values**

None

4.4.15 interrupt_disable

(1) Synopsis

Disables interrupt processing.

(2) C language format

static void interrupt_disable (void);

(3) Parameters

None

(4) Description

This function disables interrupt processing.

(5) Returned values

None

4.4.16 creat_message_header

(1) **Synopsis**

Creates sample message headers.

(2) **C language format**

static void creat_message_header (void);

(3) **Parameters**

None

(4) **Description**

This function creates sample message headers.

(5) **Returned values**

None

4.4.17 user_gl_err_callback

(1) Synopsis

Callback function (CAN global error)

(2) C language format

static void user_gl_err_callback (void);

(3) Parameters

None

(4) Description

This is the callback function called when a CAN global error occurs.

For the processing, see Figure 4-23 Handling of the Callback Function for a CAN Global Error in the Sample Code.

(5) Returned values

None

4.4.18 user_ch1_err_callback

(1) Synopsis

Callback function (CAN1 error)

(2) C language format

static void user_ch1_err_callback (void);

(3) Parameters

None

(4) Description

This is the callback function called when a CAN1 error occurs.
For the processing, see Figure 4-24 Handling of the Callback Function for a CAN1 Error in the Sample Code.

(5) Returned values

None

4.4.19 user_rx_fifo_callback

(1) Synopsis

Callback function (CAN reception FIFO interrupt)

(2) C language format

static void user_rx_fifo_callback (void);

(3) Parameters

None

(4) Description

This is the callback function called when a CAN reception FIFO interrupt is generated.

For the processing, see Figure 4-25 Handling of the Callback Function for a CAN Reception FIFO Interrupt in the Sample Code

(5) Returned values

None

4.4.20 user_ch1_tx_callback

(1) Synopsis

Callback function (CAN1 transmission interrupt)

(2) C language format

static void user_ch1_tx_callback (void);

(3) Parameters

None

(4) Description

This is the callback function called when a CAN1 transmission interrupt is generated.

For the processing, see Figure 4-26 Handling of the Callback Function for a CAN1 Transmission Interrupt in the Sample Code

(5) Returned values

None

4.4.21 user_ch1_rx_fifo_callback

(1) Synopsis

Callback function (CAN1 transmission/reception FIFO reception completion interrupt)

(2) C language format

static void user_ch1_rx_fifo_callback (void);

(3) Parameters

None

(4) Description

This is the callback function called when a CAN1 transmission/reception FIFO interrupt reception completion interrupt is generated.

For the processing, see Figure 4-27 Handling of the Callback Function for a CAN1 Transmission/Reception FIFO Reception Completion Interrupt in the Sample Code

(5) Returned values

None

4.4.22 tx_demo_buffer

(1) Synopsis

Message transmission by using a transmission buffer

(2) C language format

static void tx_demo_buffer (void);

(3) Parameters

None

(4) Description

This function performs message transmission by using a transmission buffer.

For the processing, see Figure 4-2 Message Transmission by Using the Transmission Buffer in the Sample Code.

(5) Returned values

None

4.4.23 tx_demo_fifo

(1) Synopsis

Message transmission by using a transmission/reception FIFO buffer (transmission mode)

(2) C language format

static void tx_demo_fifo (void);

(3) Parameters

None

(4) Description

This function performs message transmission by using a transmission/reception FIFO buffer (transmission mode).

For the processing, see Figure 4-3 Message Transmission by Using the Transmission/Reception FIFO Buffer (Transmission Mode) in the Sample Code.

(5) Returned values

None

4.4.24 rx_demo_buffer

(1) Synopsis

Message reception by using a reception buffer

(2) C language format

static void rx_demo_buffer (void);

(3) Parameters

None

(4) Description

This function performs message reception by using a reception buffer.
For the processing, see Figure 4-6 Message Reception by Using the Reception Buffer in the Sample Code.

(5) Returned values

None

4.4.25 rx_demo_fifo

(1) Synopsis

Message reception by using a transmission/reception FIFO buffer (reception mode)

(2) C language format

static void rx_demo_fifo (void);

(3) Parameters

None

(4) Description

This function performs message reception by using a transmission/reception FIFO buffer (reception mode).

For the processing, see Figure 4-8 Message Reception by Using the Transmission/Reception FIFO Buffer (Reception Mode) in the Sample Code.

(5) Returned values

None

4.4.26 rx_demo_rx_fifo

(1) Synopsis

Message reception by using a reception FIFO buffer

(2) C language format

static void rx_demo_rx_fifo (void);

(3) Parameters

None

(4) Description

This function performs message reception by using a reception FIFO buffer.

For the processing, see Figure 4-7 Message Reception by Using the Reception FIFO Buffer in the Sample Code.

(5) Returned values

None

4.4.27 trx_demo_fifo

(1) **Synopsis**

Message transmission test while receiving a message

(2) **C language format**

static void trx_demo_fifo (void);

(3) **Parameters**

None

(4) **Description**

This function performs message transmission test while receiving a message.
For the processing, see Figure 4-12 Test to send a message while receiving sample code message.

(5) **Returned values**

None

4.4.28 selftest_buf_to_buf

(1) Synopsis

Test which sends a message by using the transmission buffer and receives the message by the reception buffer

(2) C language format

static void selftest_buf_to_buf (void);

(3) Parameters

None

(4) Description

This function carries out a test which sends a message by using the transmission buffer and receives the message by the reception buffer.

For the processing, see Figure 4-15 and Figure 4-16 Test which sends a message by using the transmission buffer and receives the message by the reception buffer in the Sample Code.

(5) Returned values

None

4.4.29 selftest_buf_to_rx_fifo

(1) Synopsis

Test which sends a message by using the transmission buffer and receives the message by the reception FIFO buffer

(2) C language format

static void selftest_buf_to_rx_fifo (void);

(3) Parameters

None

(4) Description

This function carries out a test which sends a message by using the transmission buffer and receives the message by the reception FIFO buffer.

For the processing, see Figure 4-17 and Figure 4-18 Test Which Sends a Message by Using the Transmission Buffer and Receives the Message by the Reception FIFO Buffer in the Sample Code.

(5) Returned values

None

4.4.30 selftest_buf_to_fifo

(1) Synopsis

Test which sends a message by using the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode)

(2) C language format

static void selftest_buf_to_fifo (void);

(3) Parameters

None

(4) Description

This function carries out a test which sends a message by using the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode).

For the processing, see Figure 4-19 and Figure 4-20 Test Which Sends a Message by Using the Transmission Buffer and Receives the Message by the Transmission/Reception FIFO Buffer (Reception Mode) in the Sample Code.

(5) Returned values

None

4.4.31 selftest_fifo_to_fifo

(1) Synopsis

Test which sends a message by using the transmission/reception FIFO buffer (transmission mode) and receives the message by the transmission/reception FIFO buffer (reception mode)

(2) C language format

static void selftest_fifo_to_fifo (void);

(3) Parameters

None

(4) Description

This function carries out a test which sends a message by using the transmission/reception FIFO buffer (transmission mode) and receives the message by the transmission/reception FIFO buffer (reception mode).

For the processing, see Figure 4-21 and Figure 4-22 Test Which Sends a Message by Using the Transmission/Reception Buffer (Transmission Mode) and Receives the Message by the Transmission/Reception FIFO Buffer (Reception Mode) in the Sample Code.

(5) Returned values

None

4.4.32 demo_result

(1) Synopsis

Outputs the result of processing.

(2) C language format

static void demo_result (void);

(3) Parameters

None

(4) Description

This function outputs the result of the test.

(5) Returned values

None

4.4.33 demo_init

(1) **Synopsis**

Initializes the variables and others to be used.

(2) **C language format**

static void demo_init (void);

(3) **Parameters**

None

(4) **Description**

This function initializes the variables and others to be used.

(5) **Returned values**

None

4.4.34 demo_end

(1) Synopsis

Clears the information about the buffer used.

(2) C language format

static void demo_end(void);

(3) Parameters

None

(4) Description

This function clears the information about the buffer used.

(5) Returned values

None

4.4.35 rx_rule

(1) Synopsis

Sets CAN reception rules.

(2) C language format

static void rx_rule(uint32_t buf_type);

(3) Parameters

I/O	Parameter		Description
I	uint32_t	buf_type	Buffer type

(4) Description

This function is used for setting CAN reception rules.

The information of the reception rules is stored in the can_rx_rule_t structure and settings are made in R_CAN_RxSet().

For details, refer to section 27.5.1, Data Processing Using the Reception Rule Table and section 27.9.1.4, Reception Rule Setting, in the EC-1 User's Manual: Hardware.

(5) Returned values

None

4.4.36 send_termination_code

(1) Synopsis

Sends a message termination code.

(2) C language format

static void send_termination_code(void);

(3) Parameters

None

(4) Description

This function sends a message termination code.

(5) Returned values

None

4.4.37 output_tx_msg_format

(1) **Synopsis**

Outputs the format of the messages for transmission.

(2) **C language format**

static void output_tx_msg_format(void);

(3) **Parameters**

None

(4) **Description**

This function displays the format of the messages for transmission.

(5) **Returned values**

None

4.4.38 output_tx_msg_data

(1) Synopsis

Outputs the messages for transmission.

(2) C language format

static void output_tx_msg_data(uint32_t offset, uint32_t num);

(3) Parameters

I/O	Parameter		Description
I	uint32_t	offset	Transmission offset for the message buffer
I	uint32_t	num	Number of messages to be sent

(4) Description

This function sends the messages specified in the arguments.

(5) Returned values

None

4.4.39 output_rx_msg_data

(1) Synopsis

Outputs the received messages.

(2) C language format

static void output_rx_msg_data(uint32_t offset, uint32_t num)

(3) Parameters

I/O	Parameter		Description
I	uint32_t	offset	Reception offset for the message buffer
I	uint32_t	num	Number of received messages

(4) Description

This function outputs the received messages specified in the arguments.

(5) Returned values

None

4.4.40 output_rx_msg_format

(1) Synopsis

Outputs the format of the received messages.

(2) C language format

static void output_rx_msg_format (void)

(3) Parameters

None

(4) Description

This function outputs the format of the received messages.

(5) Returned values

None

4.4.41 clear_status

(1) Synopsis

Clears the flags used in the sample program.

(2) C language format

static void clear_status(void);

(3) Parameters

None

(4) Description

This function clears the following flags used in the sample program:

- Global error flag
- Channel error flag
- Transmission completion flag
- Transmission/reception FIFO flag
- Reception FIFO flag

It also initializes the buffer which stores the received messages.

(5) Returned values

None

4.4.42 write_buffer

(1) Synopsis

Writes messages for transmission to the register related to the transmission buffer.

(2) C language format

static uint32_t write_buffer(uint32_t msg_type, tx_data_t * obj);

(3) Parameters

I/O	Parameter		Description
I	uint32_t	msg_type	Message type
I	tx_data_t	* obj	Pointer to the information for transmission

(4) Description

This function writes messages for transmission to the register related to the transmission buffer.

For the processing, see Figure 4-4 Writing of Messages for Transmission to the Registers Related to the Transmission Buffer in the Sample Code.

(5) Returned values

Returned Value	Meaning
RET_OK	Transmission completion
RET_ERROR	Error
RET_BUSY	The transmission buffers are in use.

4.4.43 write_fifo

(1) Synopsis

Writes messages for transmission to the register related to the transmission/reception FIFO buffer.

(2) C language format

static uint32_t write_fifo(uint32_t msg_type, tx_data_t * obj);

(3) Parameters

I/O	Parameter		Description
I	uint32_t	msg_type	Message type
I	tx_data_t	* obj	Pointer to the information for transmission

(4) Description

This function writes messages for transmission to the register related to the transmission/reception FIFO buffer.

For the processing, see Figure 4-5 Writing of Messages for Transmission to the Registers Related to the Transmission/Reception FIFO Buffer in the Sample Code.

(5) Returned values

Returned Value	Meaning
RET_OK	Transmission completion
RET_ERROR	Error
RET_BUSY	The transmission buffers are in use.

4.4.44 read_buffer

(1) Synopsis

Reads received messages from the reception buffer.

(2) C language format

static uint32_t read_buffer(rx_data_t * obj);

(3) Parameters

I/O	Parameter	Description
I	rx_data_t * obj	Pointer to the received data storage information

(4) Description

This function reads received messages from the reception buffer.

For the processing, see Figure 4-9 Reading of Received Messages from the Reception Buffer in the Sample Code.

(5) Returned values

Returned Value	Meaning
MSG_NONE	Not received yet
MSG_BODY	Message data
MSG_END	Delimiter code

4.4.45 read_rx_fifo

(1) Synopsis

Reads received messages from the transmission/reception FIFO buffer.

(2) C language format

static void read_rx_fifo(rx_data_t * obj);

(3) Parameters

I/O	Parameter	Description
I	rx_data_t * obj	Pointer to the received data storage information

(4) Description

This function reads received messages from the reception FIFO buffer.

For the processing, see Figure 4-10 Reading of Received Messages from the Reception FIFO Buffer in the Sample Code.

(5) Returned values

None

4.4.46 read_fifo

(1) Synopsis

Reads received messages from the transmission/reception FIFO buffer.

(2) C language format

static void read_fifo(uint32_t ch, rx_data_t * obj);

(3) Parameters

I/O	Parameter		Description
I	uint32_t	ch	Channel
I	rx_data_t	* obj	Pointer to the received data storage information

(4) Description

This function reads received messages from the reception FIFO buffer.

For the processing, see Figure 4-11 Reading of Received Messages from the Transmission/Reception FIFO Buffer in the Sample Code.

(5) Returned values

None

4.4.47 set_fifo_buffer

(1) Synopsis

Registers the transmission/reception FIFO buffer.

(2) C language format

static void set_fifo_buffer(uint32_t ch, uint32_t mode, uint32_t condition);

(3) Parameters

I/O	Parameter		Description
I	uint32_t	ch	Channel
I	uint32_t	mode	FIFO buffer mode
I	uint32_t	condition	Trigger condition

(4) Description

This function registers the transmission/reception FIFO buffer.

(5) Returned values

None

4.4.48 rx_fifo_mode

(1) Synopsis

Registers the reception FIFO buffer.

(2) C language format

static void rx_fifo_mode(uint32_t ch, uint32_t condition);

(3) Parameters

I/O	Parameter		Description
I	uint32_t	ch	Channel
I	uint32_t	condition	Trigger condition

(4) Description

This function registers the reception FIFO buffer.

(5) Returned values

None

4.4.49 get_error_status

(1) **Synopsis**

Gets the interrupt source.

(2) **C language format**

static void get_error_status(void);

(3) **Parameters**

None

(4) **Description**

This function gets each interrupt source by using the R_CAN_GetInterruptSource () function.

(5) **Returned values**

None

4.5 Flowcharts

4.5.1 Main Processing

This sample program allows the user to select a check item from the menu.

For the menu, see section 4.6.5, Functions of the Sample Program in section 4.6, Tutorials.

Figure 4-1 is a flowchart for the main processing in the sample code.

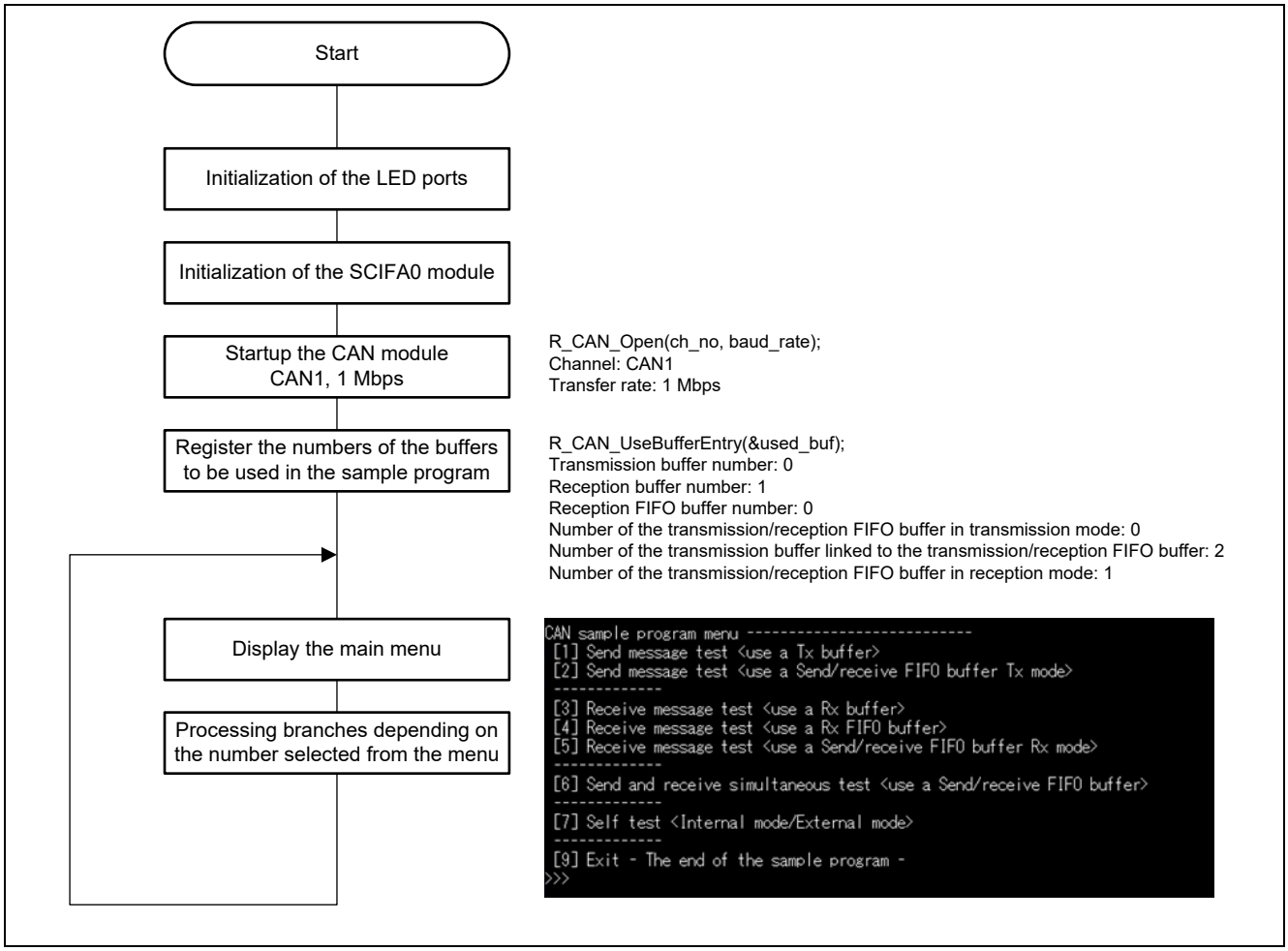


Figure 4-1 Main Processing in the Sample Code

4.5.2 Transmission Test

There are two types of transmission tests: message transmission by using a transmission buffer and message transmission by using a transmission/reception FIFO buffer in transmission mode, which can be selected from the menu.

For the menu, see section 4.6.5, Functions of the Sample Program in section 4.6, Tutorials.

The following functions are used to perform each test.

- void tx_demo_buffer(void)
  Message transmission by using the transmission buffer
- void tx_demo_fifo(void)
  Message transmission by using the transmission/reception FIFO buffer in transmission mode
- uint32_t write_buffer(uint32_t msg_type, tx_data_t * obj)
  Writing messages for transmission to the registers related to the transmission buffer
- uint32_t write_fifo(uint32_t msg_type, tx_data_t * obj)
  Writing messages for transmission to the registers related to the transmission/reception FIFO buffer

Figure 4-2 to Figure 4-5 show the flowcharts for processing by the respective functions.

- Message transmission by using the transmission buffer

  Function name: void tx_demo_buffer(void)

  The figure below is a flowchart showing the message transmission by using the transmission buffer.

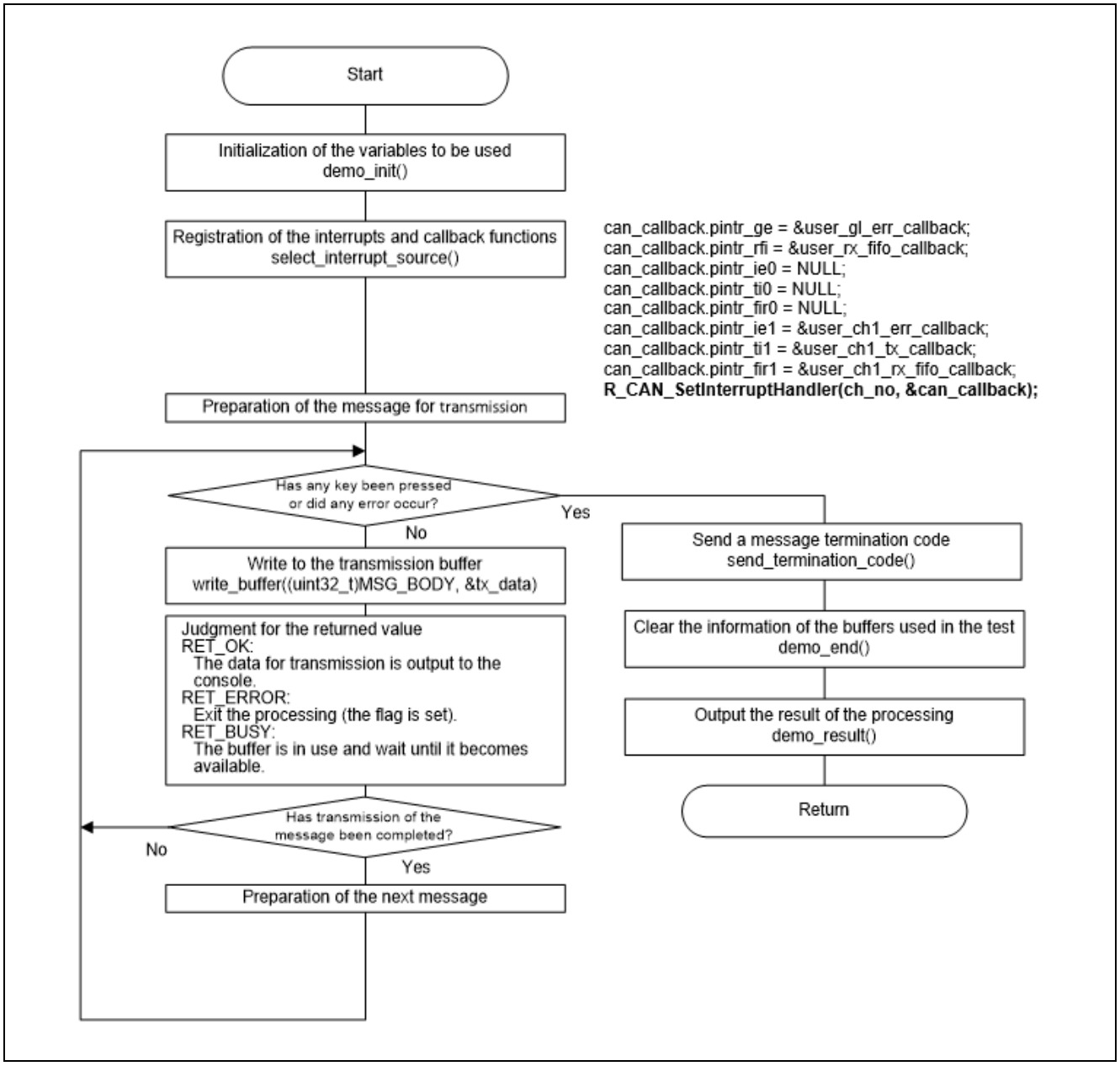


Figure 4-2 Message Transmission by Using the Transmission Buffer in the Sample Code

- Message transmission by using the transmission/reception FIFO buffer in transmission mode

  Function name: void tx_demo_fifo(void)

  The figure below is a flowchart showing the message transmission by using the transmission/reception FIFO buffer in transmission mode.

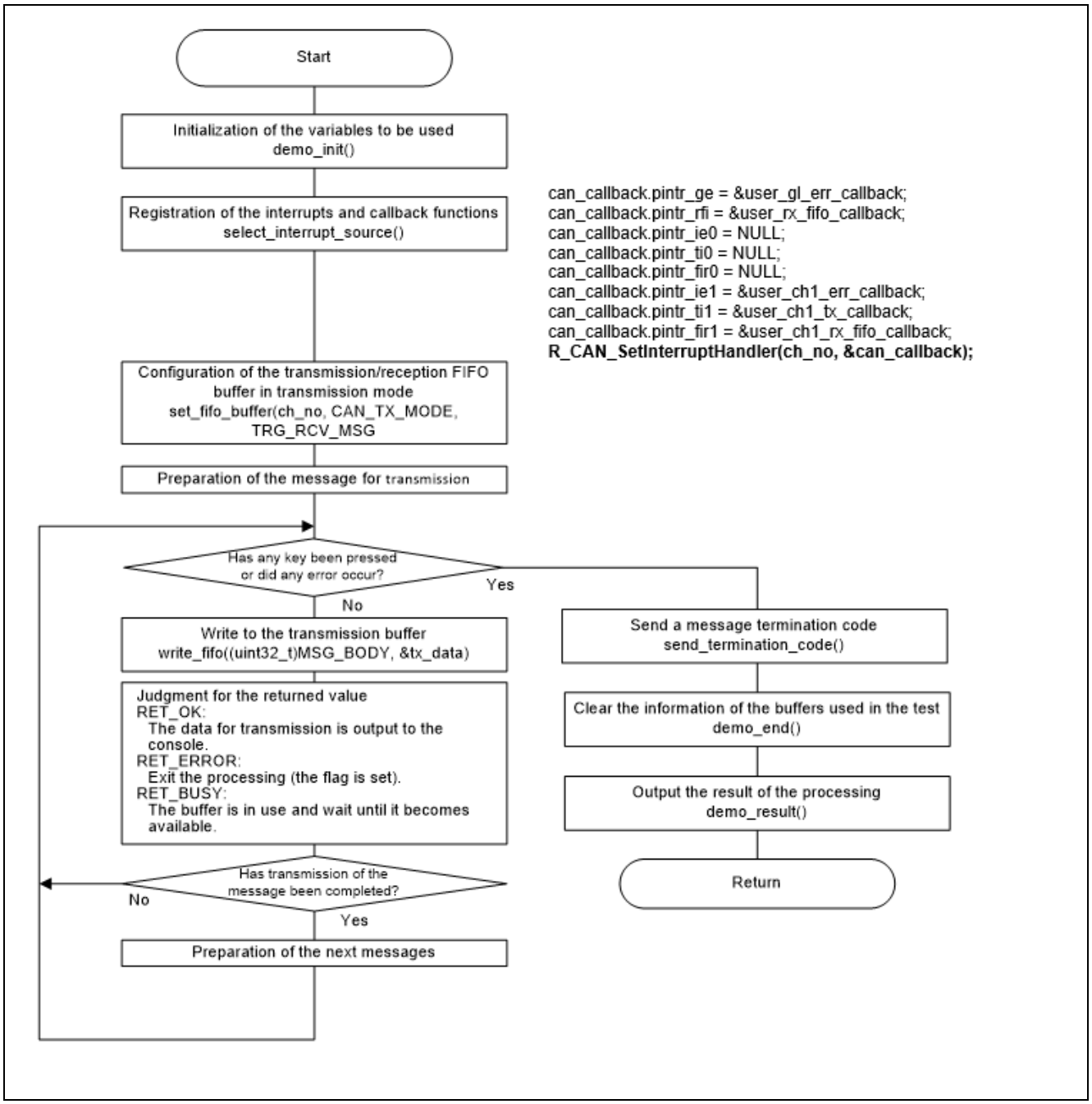


Figure 4-3 Message Transmission by Using the Transmission/Reception FIFO Buffer (Transmission Mode) in the Sample Code

- Writing messages for transmission to the registers related to the transmission buffer

  Function name: uint32_t write_buffer(uint32_t msg_type, tx_data_t * obj)

  The figure below is a flowchart showing the writing of messages for transmission to the registers related to the transmission buffer.

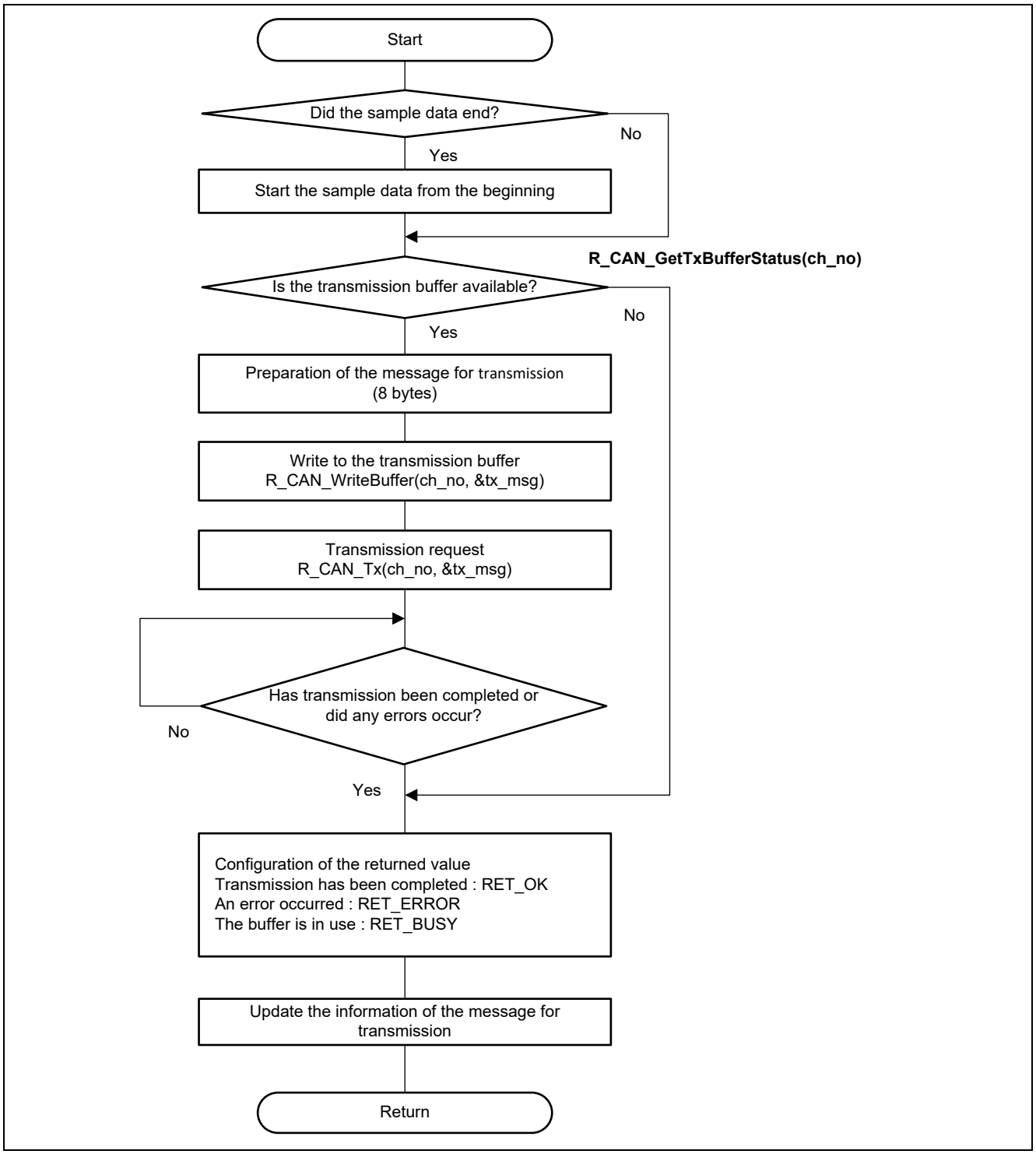


Figure 4-4 Writing of Messages for Transmission to the Registers Related to the Transmission Buffer in the Sample Code

- Writing messages for transmission to the registers related to the transmission/reception FIFO buffer

  Function name: uint32_t write_fifo(uint32_t msg_type, tx_data_t * obj)

  The figure below is a flowchart showing the writing of messages for transmission to the registers related to the transmission/reception FIFO buffer.

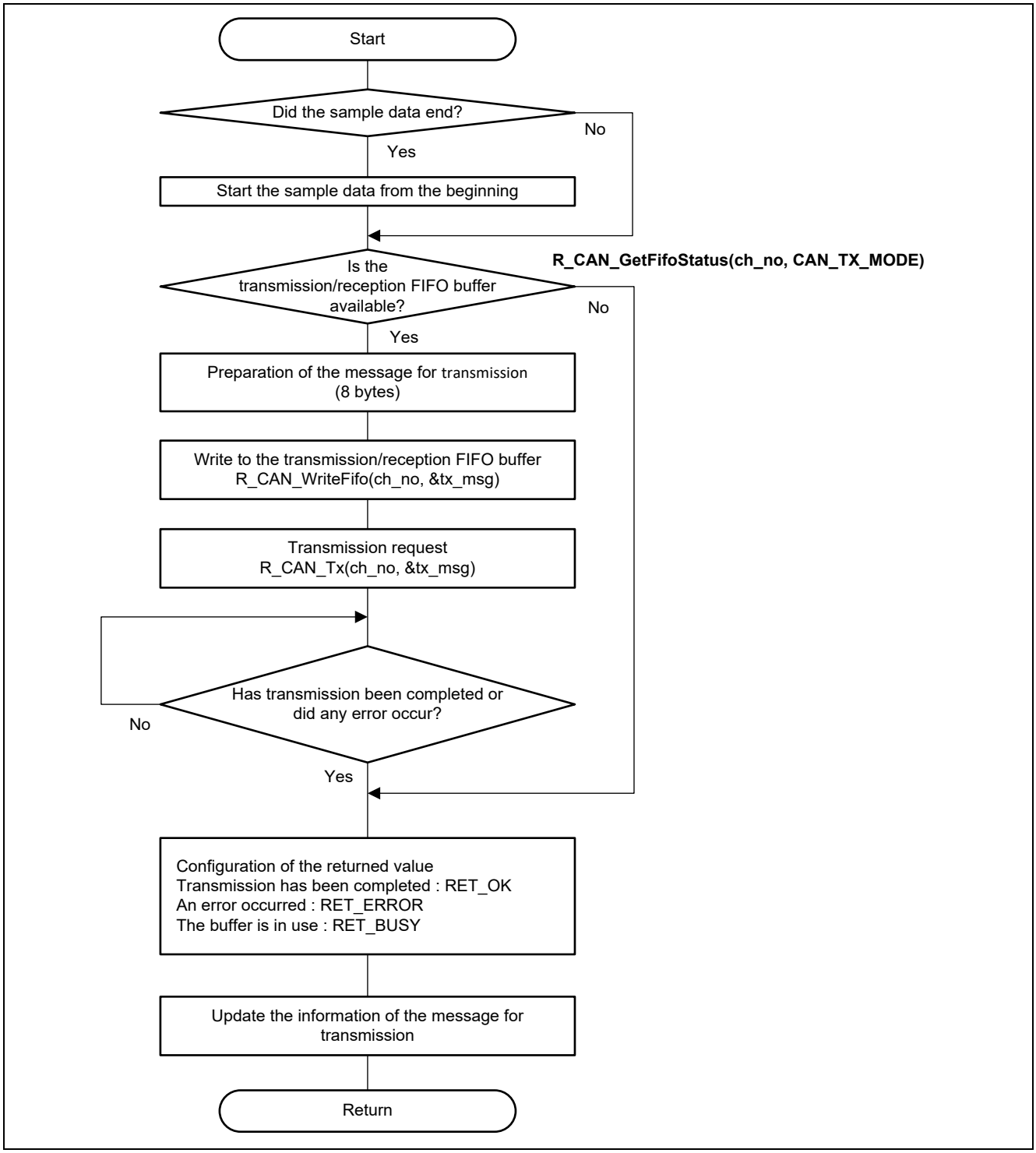


Figure 4-5 Writing of Messages for Transmission to the Registers Related to the Transmission/Reception FIFO Buffer in the Sample Code

4.5.3 Reception Test

There are three types of reception tests: message reception by using the reception buffer, message reception by using the reception FIFO buffer and message reception by using the transmission/reception FIFO buffer in reception mode, which can be selected from the menu.

For the menu, see section 4.6.5, Functions of the Sample Program in section 4.6, Tutorials.

The following functions are used to perform each test.

- void rx_demo_buffer(void)
  Message reception by using the reception buffer
- void rx_demo_rx_fifo(void)
  Message reception by using the reception FIFO buffer
- void rx_demo_fifo(void)
  Message reception by using the transmission/reception FIFO buffer in reception mode
- uint32_t read_buffer(rx_data_t * obj)
  Reading received messages from the reception buffer
- void read_rx_fifo(rx_data_t * obj)
  Reading received messages from the reception FIFO buffer
- void read_fifo(uint32_t ch, rx_data_t * obj)
  Reading received messages from the transmission/reception FIFO buffer

Figure 4-6 to Figure 4-11 show the flowcharts for processing by the respective functions.

- Message reception by using the reception buffer

  Function name: void rx_demo_buffer(void)

  The figure below is a flowchart showing the message reception by using the reception buffer.

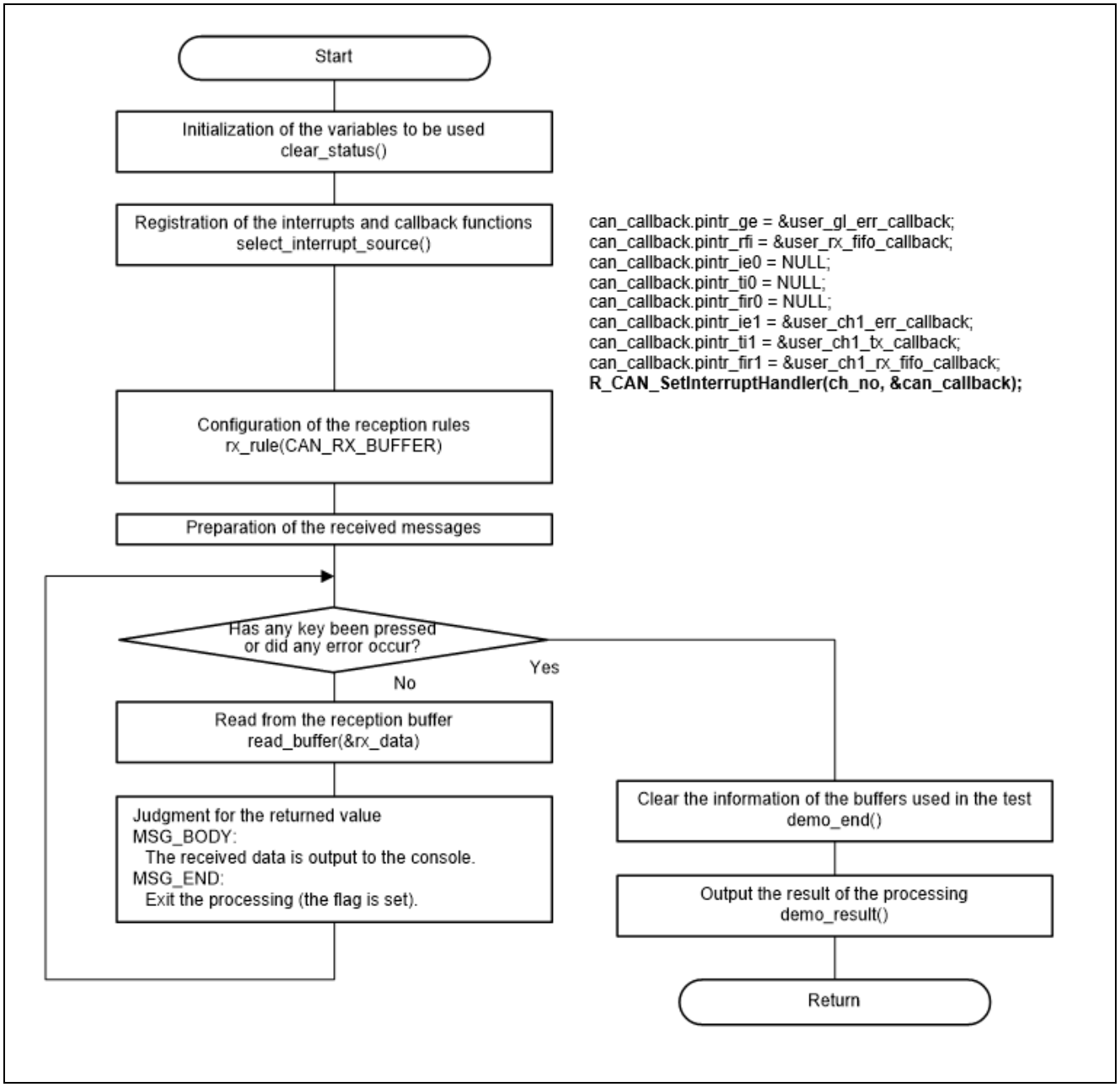


Figure 4-6 Message Reception by Using the Reception Buffer in the Sample Code

- Message reception by using a reception FIFO buffer

  Function name: void rx_demo_rx_fifo(void)

  The figure below is a flowchart showing the message reception by using the reception FIFO buffer.

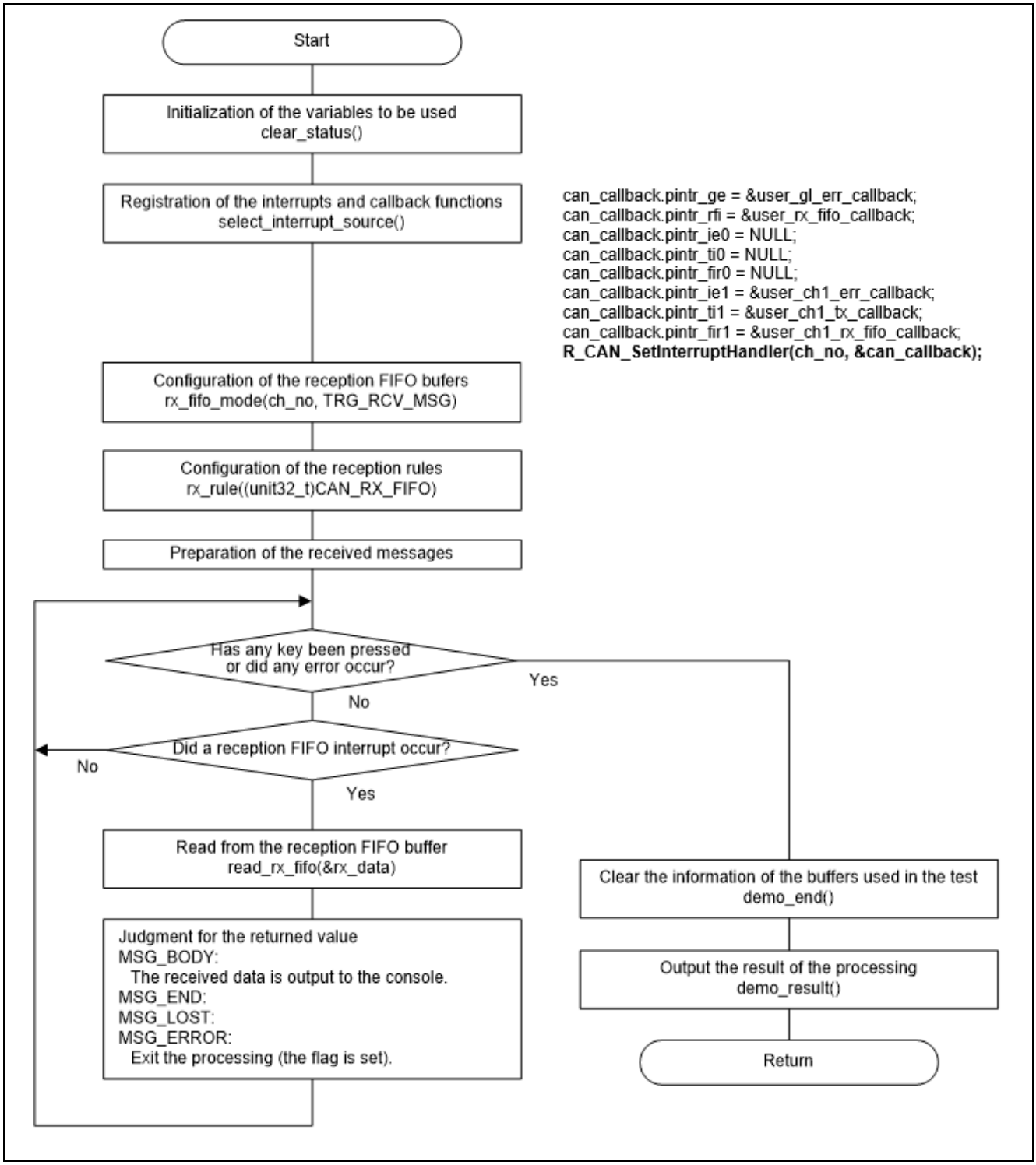


Figure 4-7 Message Reception by Using the Reception FIFO Buffer in the Sample Code

- Message reception by using the transmission/reception FIFO buffer in reception mode

  Function name: void rx_demo_fifo(void)

  The figure below is a flowchart showing the message reception by using the transmission/reception FIFO buffer in reception mode.

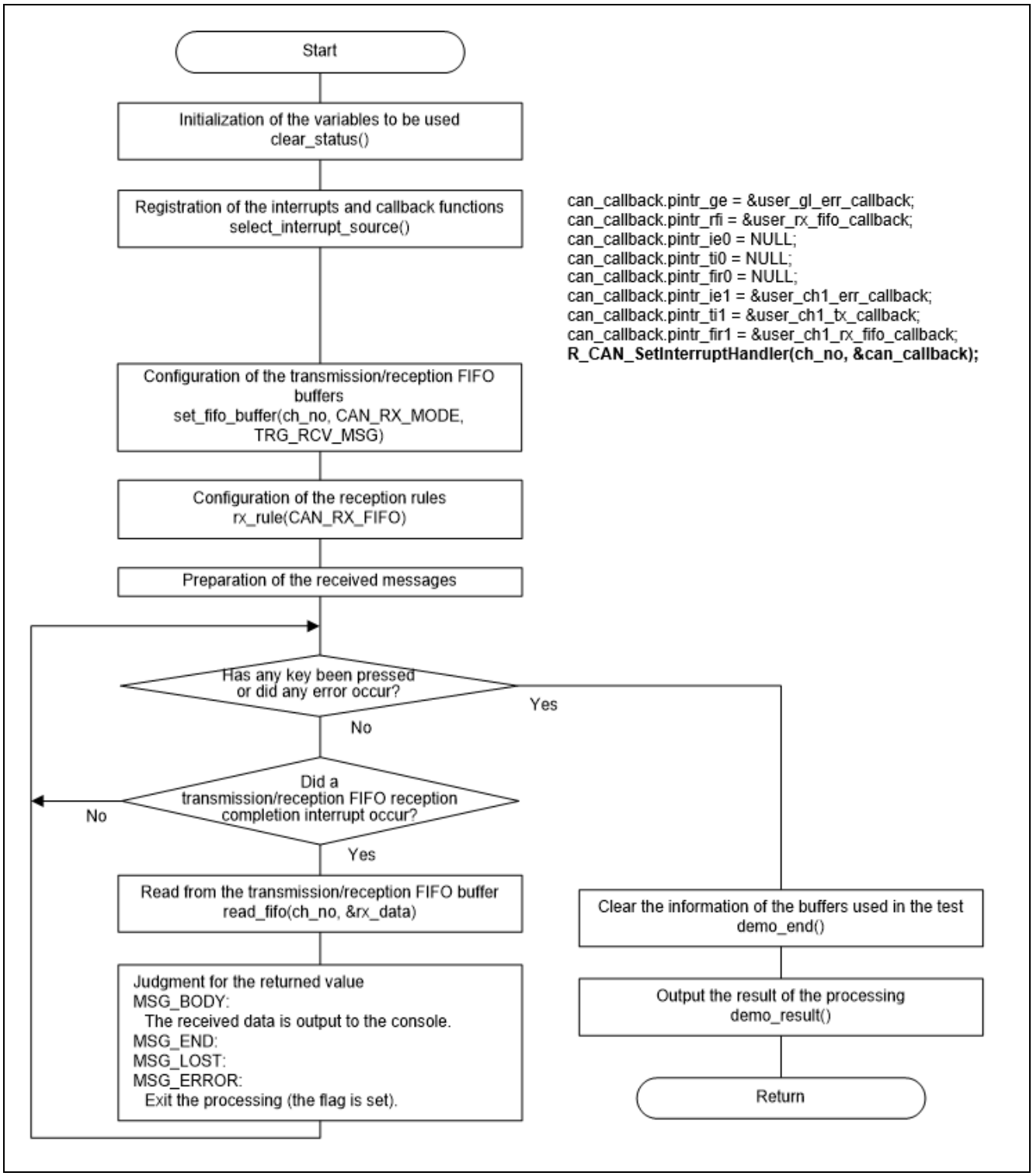


Figure 4-8 Message Reception by Using the Transmission/Reception FIFO Buffer (Reception Mode) in the Sample Code

- Reading received messages from the reception buffer

  Function name: uint32_t read_buffer(rx_data_t * obj)

  The figure below is a flowchart showing the reading of received messages from the reception buffer.

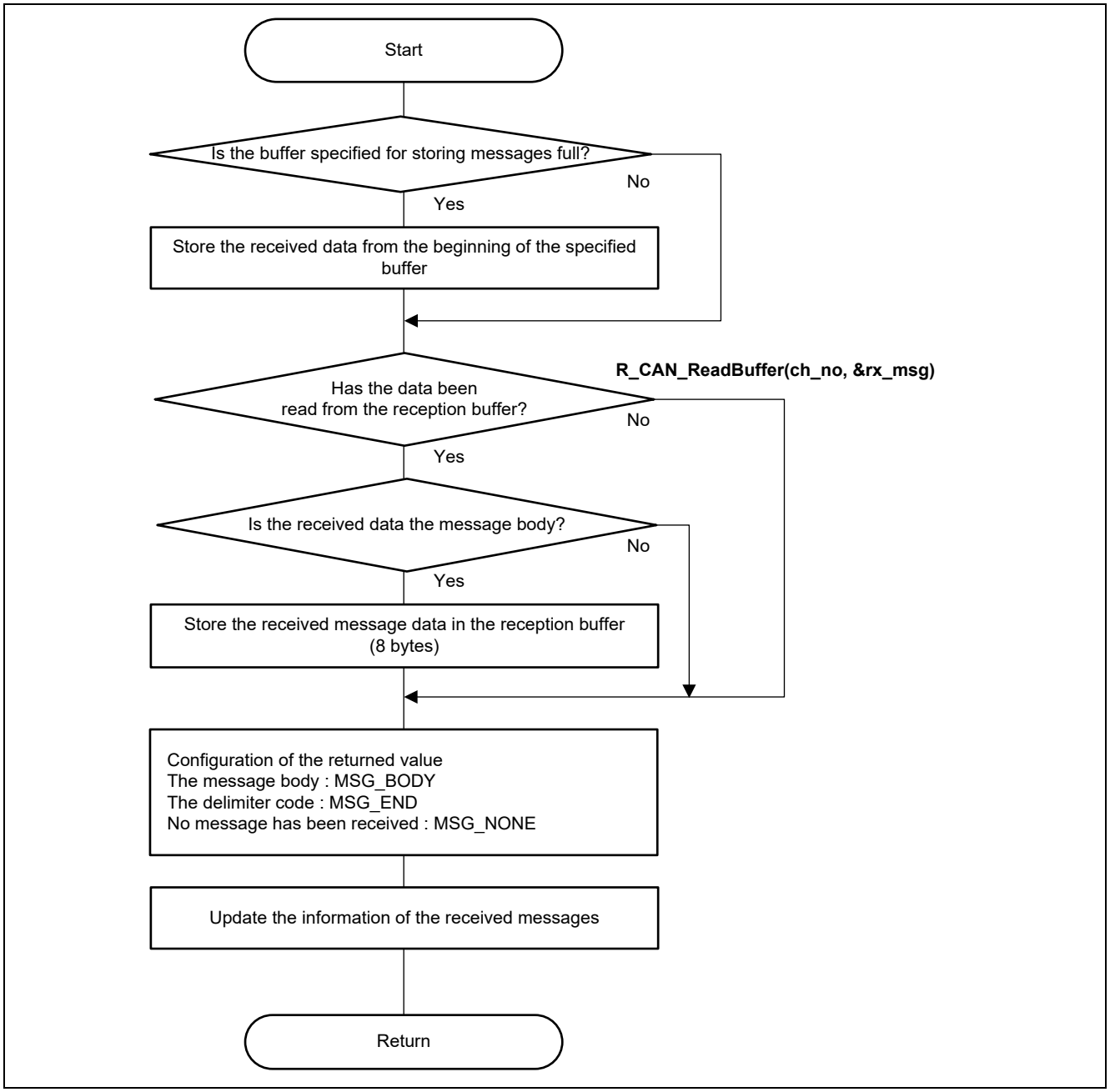


Figure 4-9 Reading of Received Messages from the Reception Buffer in the Sample Code

- Reading received messages from the reception FIFO buffer

  Function name: void read_rx_fifo(rx_data_t * obj)

  The figure below is a flowchart showing the reading of received messages from the reception FIFO buffer.

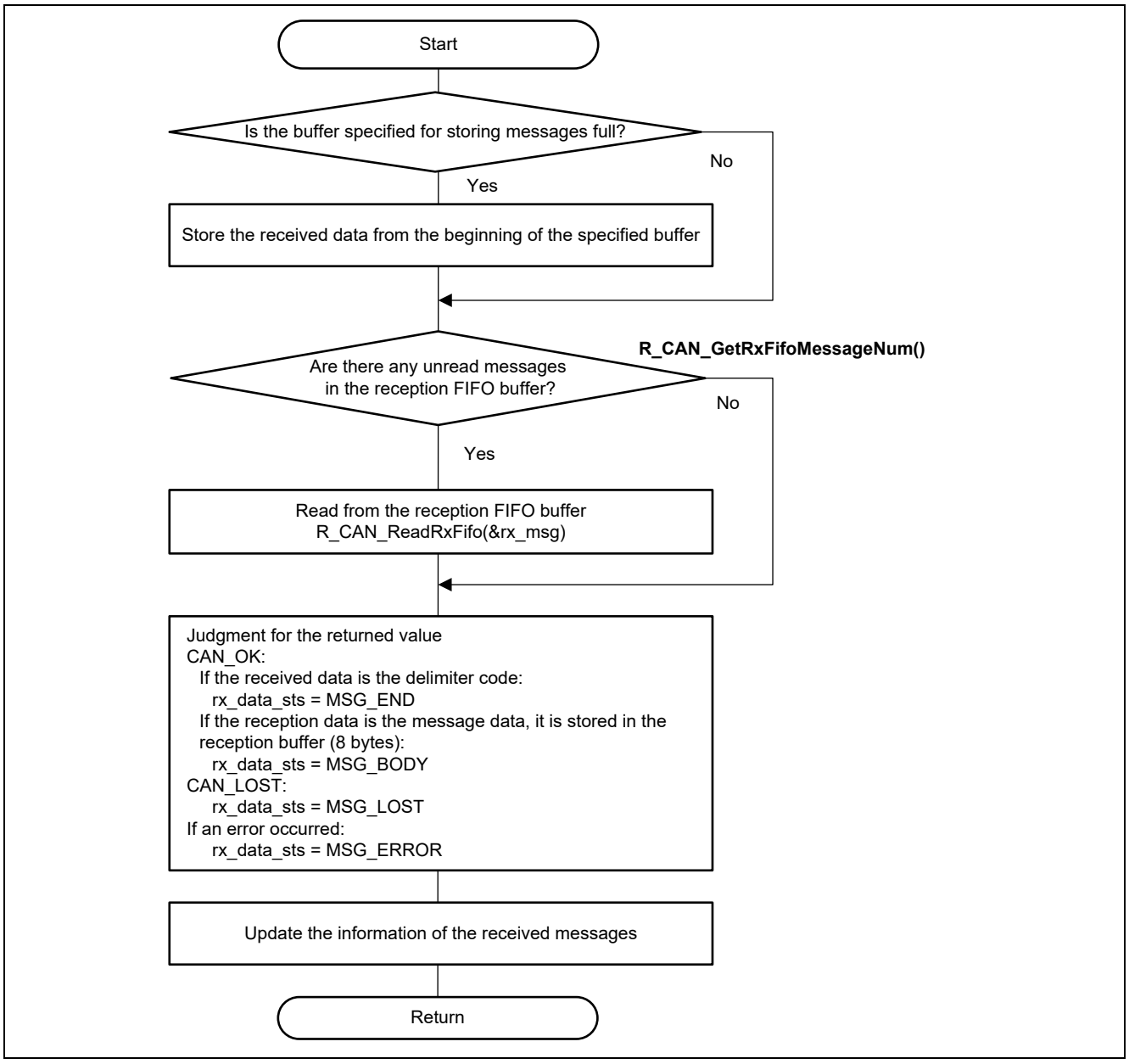


Figure 4-10 Reading of Received Messages from the Reception FIFO Buffer in the Sample Code

- Reading received messages from the transmission/reception FIFO buffer

  Function name: void read_fifo(uint32_t ch, rx_data_t * obj)

  The figure below is a flowchart showing the reading of received messages from the transmission/reception FIFO buffer.

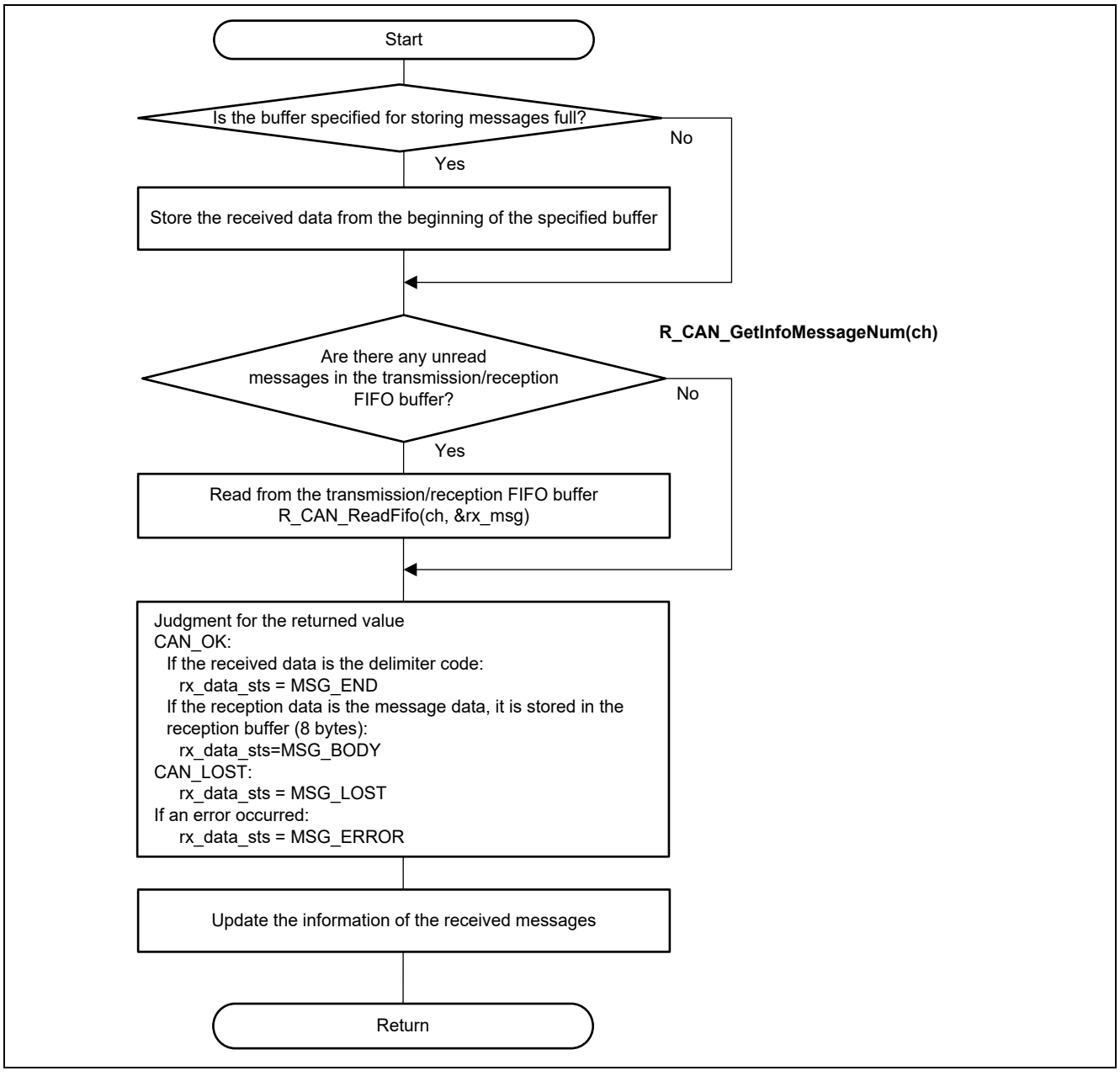


Figure 4-11 Reading of Received Messages from the Transmission/Reception FIFO Buffer in the Sample Code

4.5.4 Test for Transmission While Receiving Data at the Same Time

This is to test transmission of messages while receiving a different message at the same time.

In this sample program, the messages are received by using the transmission/reception FIFO buffer in reception mode and transmitted by using the transmission/reception FIFO buffer in transmission mode.

For the menu, see section 4.6.5, Functions of the Sample Program in section 4.6, Tutorials.

The following functions are used to perform this test.

- void trx_demo_fifo (void)
  Message transmission while receiving a different message at the same time

Figure 4-12 is a flowchart of message transmission while receiving a different message at the same time.

- Message transmission while receiving a different message at the same time (reading received messages from the transmission/reception FIFO buffer)

  Function name: void trx_demo_fifo (void)

  The figure below is a flowchart showing message transmission while receiving a different message at the same time.

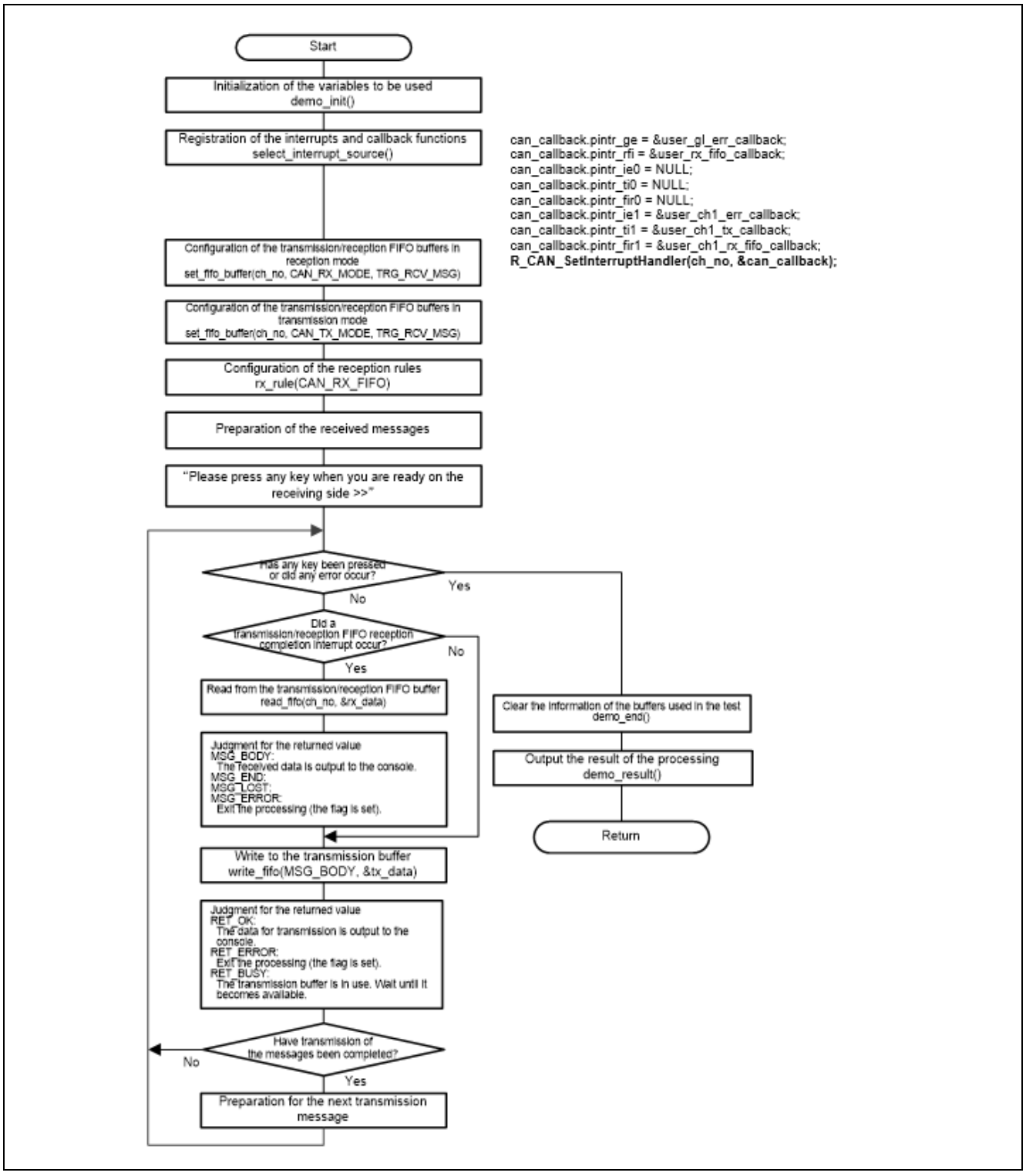


Figure 4-12 Test to send a message while receiving sample code message.

4.5.5 Self Tests

Testing CAN communications by using only the evaluation board which is connected to the development environment is possible.

In this sample program, self tests for checking transmission and reception are available.

Select the test mode from self-test mode 0 for external loopback or self-test mode 1 for internal loopback from the menu.

For the menu, see section 4.6.5, Functions of the Sample Program in section 4.6, Tutorials.

- Self-test mode 0 (external loopback mode)

  This is a loopback test mode for a channel including the CAN transceiver.

  Figure 4-13 shows the connection when self-test mode 0 is selected.

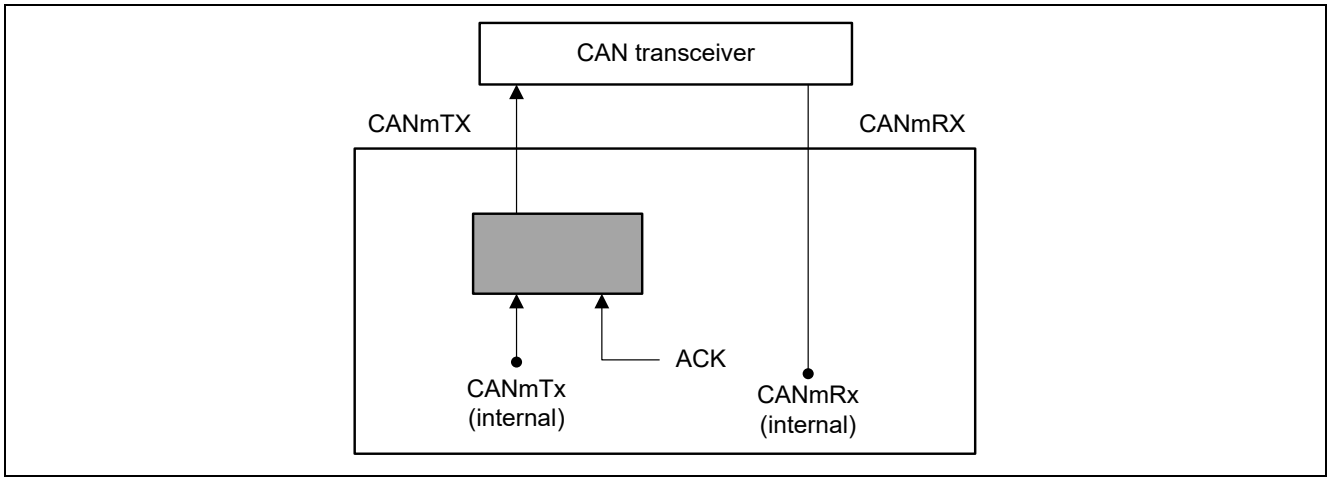


Figure 4-13 Connection for Self-Test Mode 0

- Self-test mode 1 (internal loopback mode)

  In this mode, the transmitted messages are handled as the received messages and stored in the specified buffer.

  Figure 4-14 shows the connection when self-test mode 1 is selected.

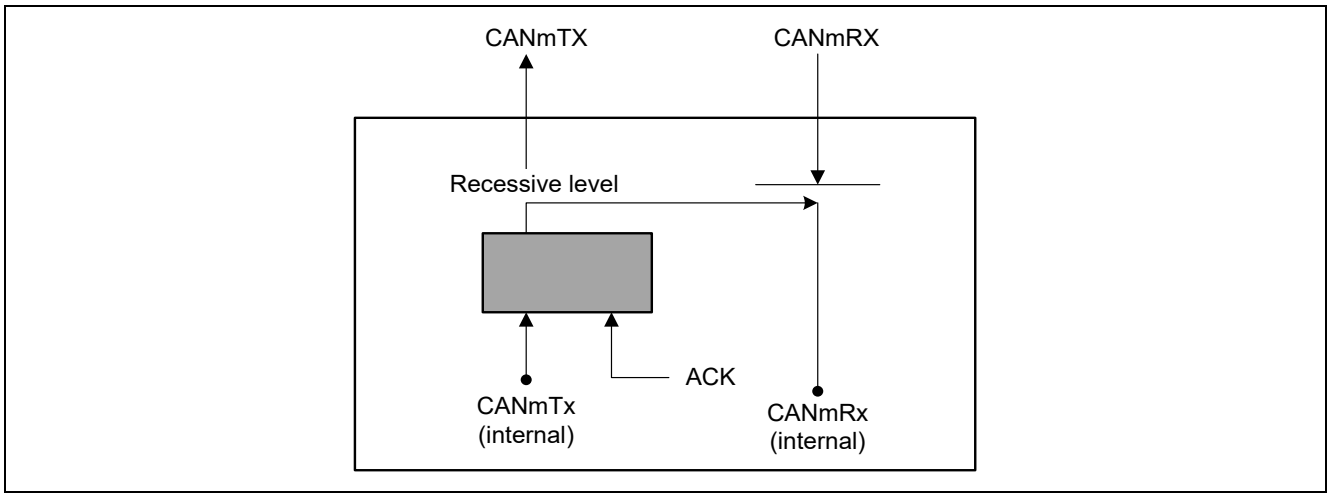


Figure 4-14 Connection for Self-Test Mode 1

Four types of self tests are available as follows:

- Test which sends a message by using the transmission buffer and receives the message by the reception buffer
- Test which sends a message by using the transmission buffer and receives the message by the reception FIFO buffer
- Test which sends a message by using the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode)
- Test which sends a message by using the transmission/reception FIFO buffer (transmission mode) and receives the message by the transmission/reception FIFO buffer (reception mode)

The type of transmission depends on the buffer used for transmission as follows:

- The transmission buffer is used:

  Transmission of one message is repeated.
- The transmission/reception FIFO buffer in transmission mode is used:

  Transmission of one message is repeated.

The type of reception depends on the buffer used for reception as follows:

- The reception FIFO buffer is used:

  An interrupt occurs when the number of messages stored in the reception FIFO buffer matches the specified FIFO buffer depth (four messages).
- The transmission/reception FIFO buffer in reception mode is used:

  An interrupt occurs when the number of messages stored in the transmission/reception FIFO buffer matches the transmission/reception FIFO buffer depth (four messages).

The following functions are used to perform each test.

- void selftest_buf_to_buf(void)

  Test which sends a message by using the transmission buffer and receives the message by the reception buffer
- void selftest_buf_to_rx_fifo(void)

  Test which sends a message by using the transmission buffer and receives the message by the reception FIFO buffer
- void selftest_buf_to_fifo(void)

  Test which sends a message by using the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode)
- void selftest_fifo_to_fifo(void)

  Test which sends a message by using the transmission/reception FIFO buffer (transmission mode) and receives the message by the transmission/reception FIFO buffer (reception mode)

Figure 4-15 to Figure 4-21 show the flowcharts for processing by the respective functions.

- Test which sends a message by using the transmission buffer and receives the message by the reception buffer

  Function name: void selftest_buf_to_buf(void)

  The figure below is a flowchart showing a test which sends a message by using the transmission buffer and receives the message by the reception buffer.

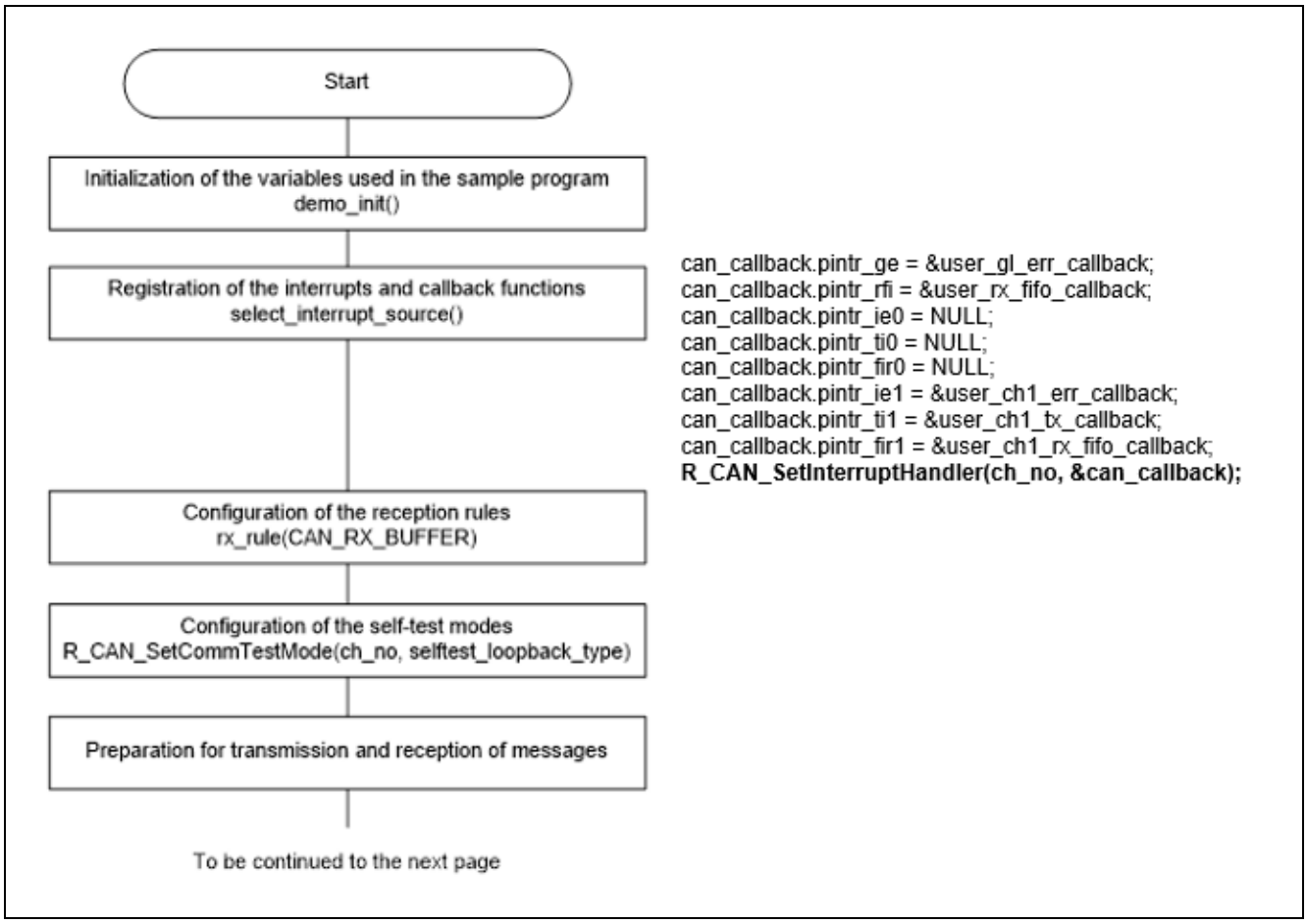


Figure 4-15 Test which sends a message by using the transmission buffer and receives the message by the reception buffer in the Sample Code. (1/2)

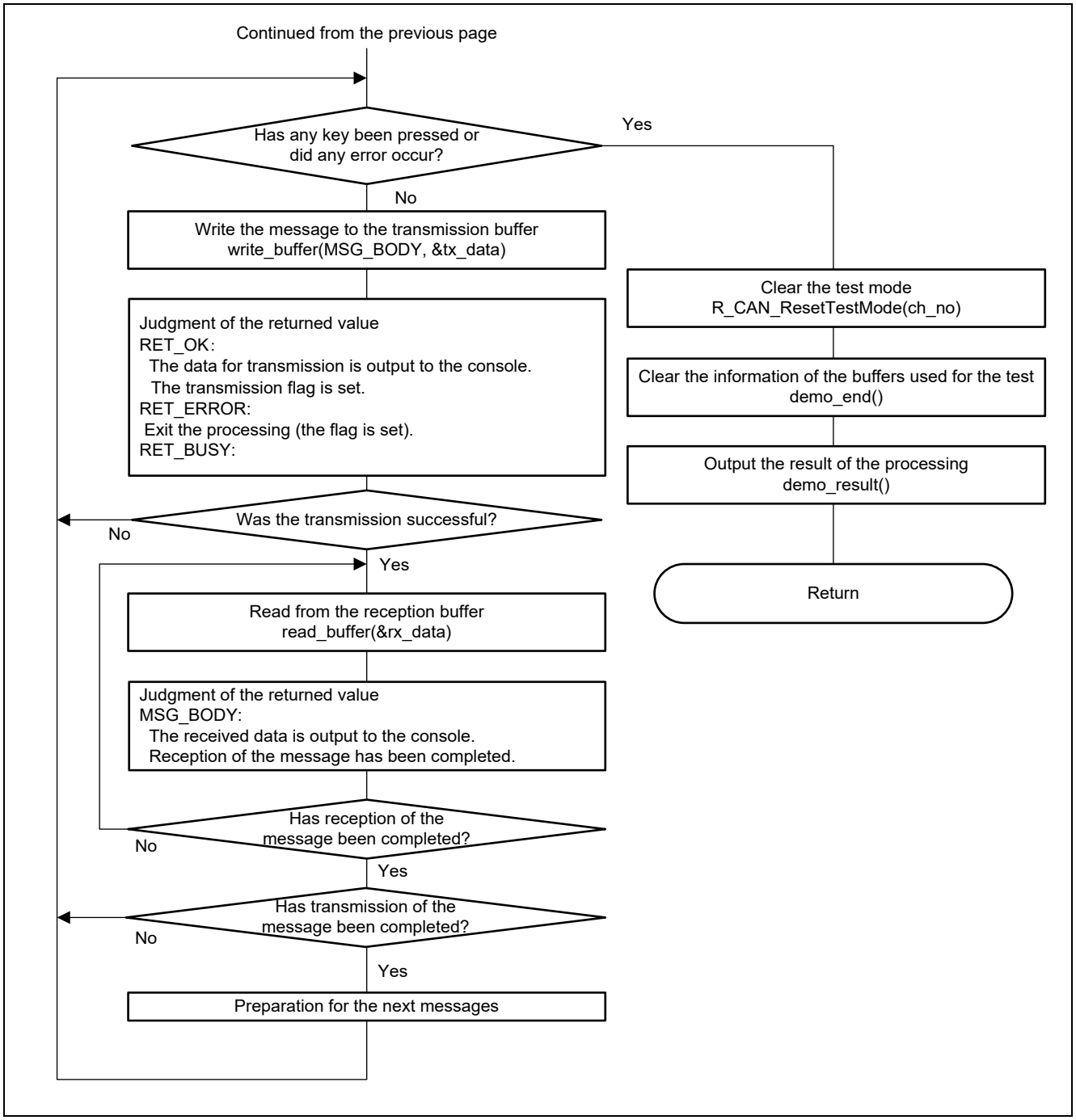


Figure 4-16 Test which sends a message by using the transmission buffer and receives the message by the reception buffer in the Sample Code. (2/2)

- Test which sends a message by using the transmission buffer and receives the message by the reception FIFO buffer

  Function name: void selftest_buf_to_rx_fifo(void)

  The figure below is a flowchart showing a test which sends a message by using the transmission buffer and receives the message by the reception FIFO buffer.

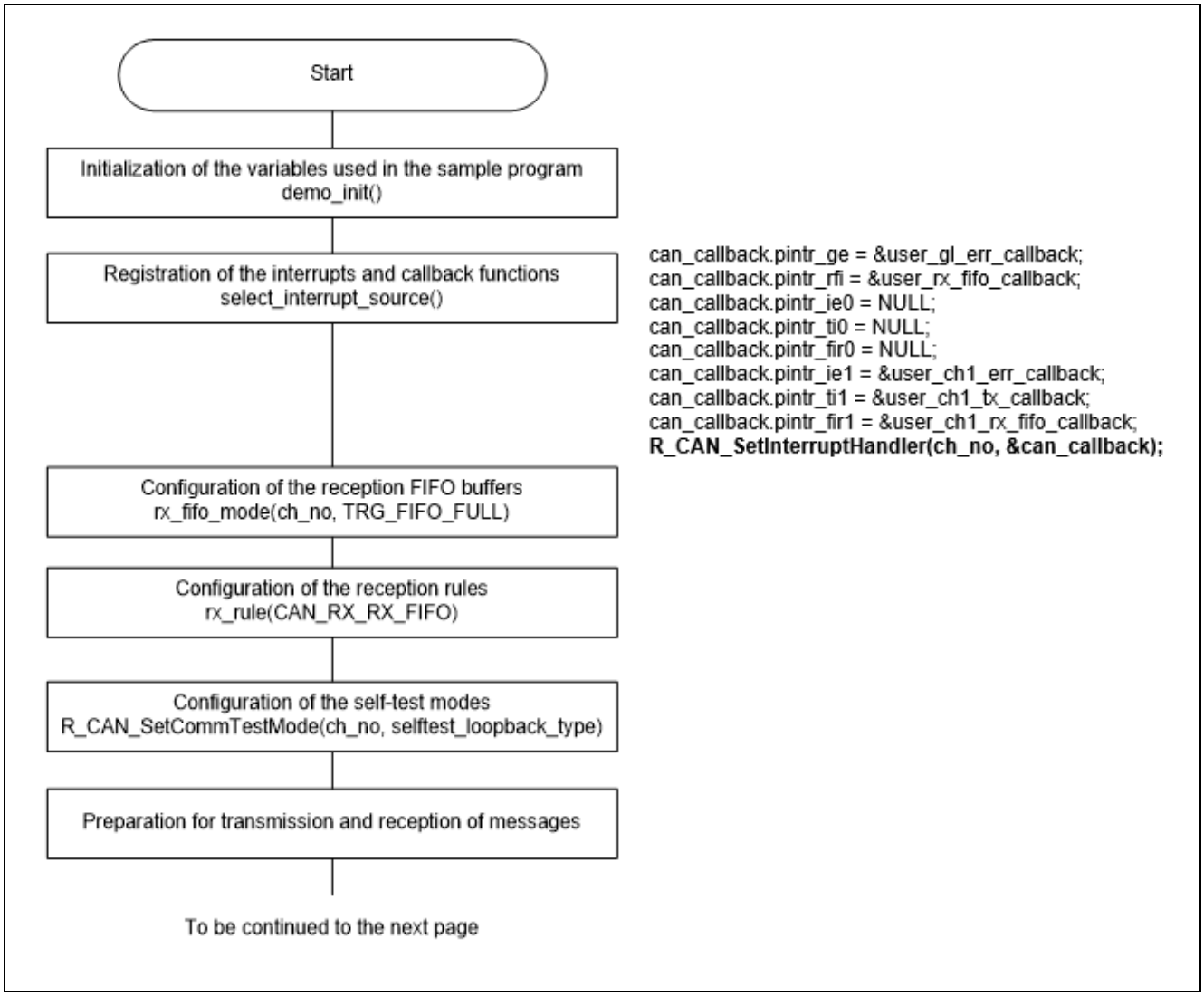


Figure 4-17 Test Which Sends a Message by Using the Transmission Buffer and Receives the Message by the Reception FIFO Buffer in the Sample Code (1/2)

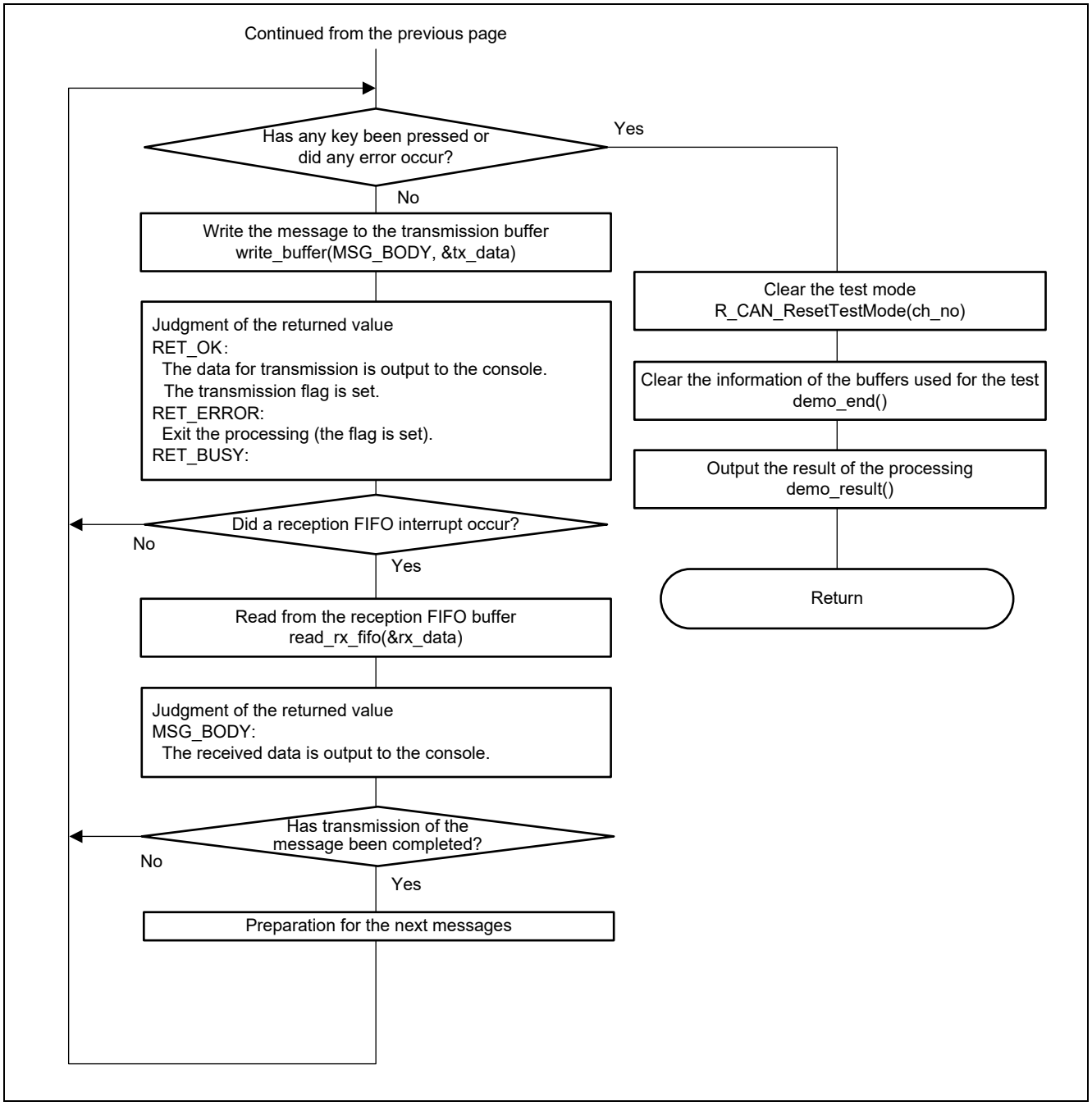


Figure 4-18 Test Which Sends a Message by Using the Transmission Buffer and Receives the Message by the Reception FIFO Buffer in the Sample Code. (2/2)

- Test which sends a message by using the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode)

  Function name: void selftest_buf_to_fifo(void)

  The figure below is a flowchart showing a test which sends a message by using the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode).

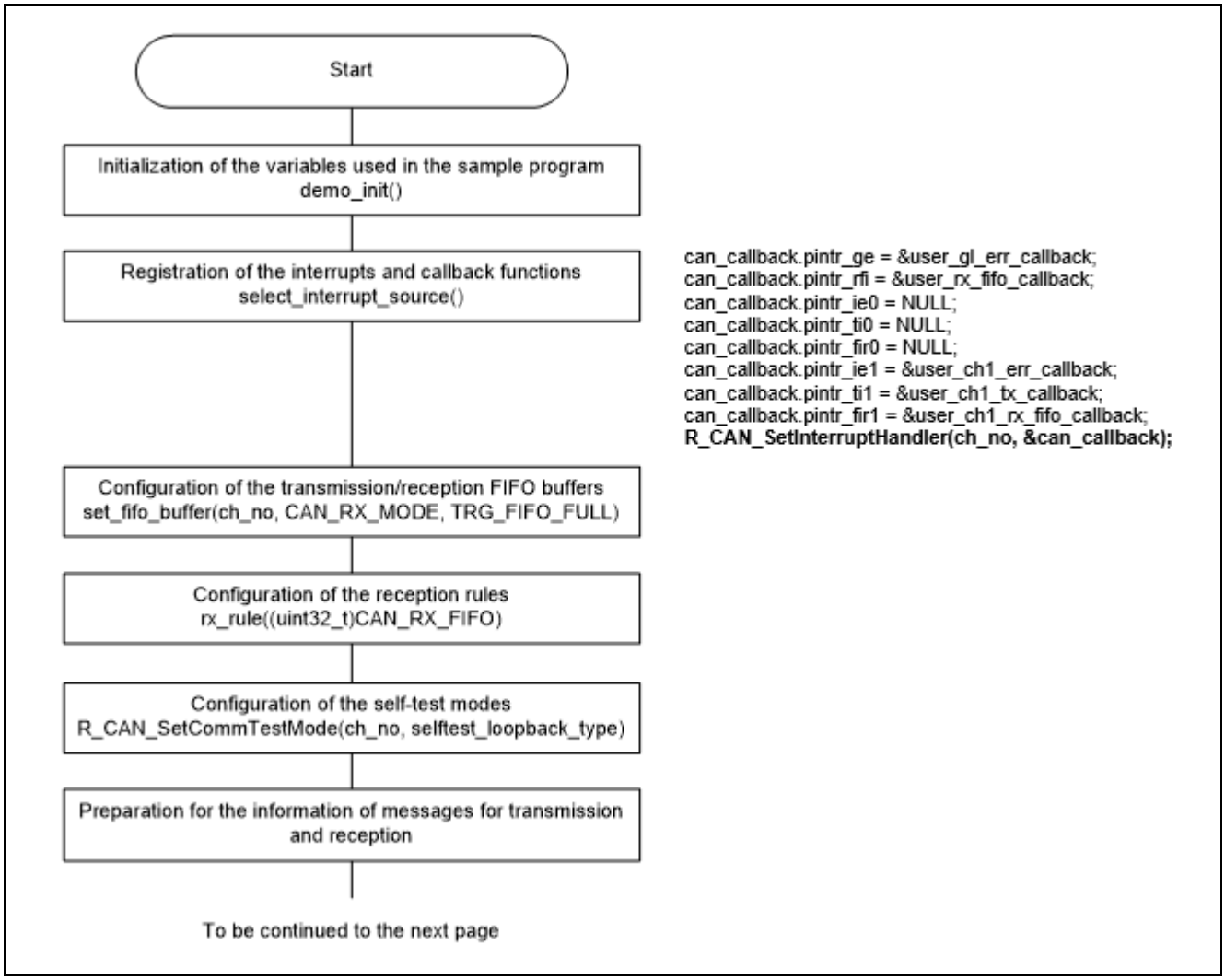


Figure 4-19 Test Which Sends a Message by Using the Transmission Buffer and Receives the Message by the Transmission/Reception FIFO Buffer (Reception Mode) in the Sample Code (1/2)

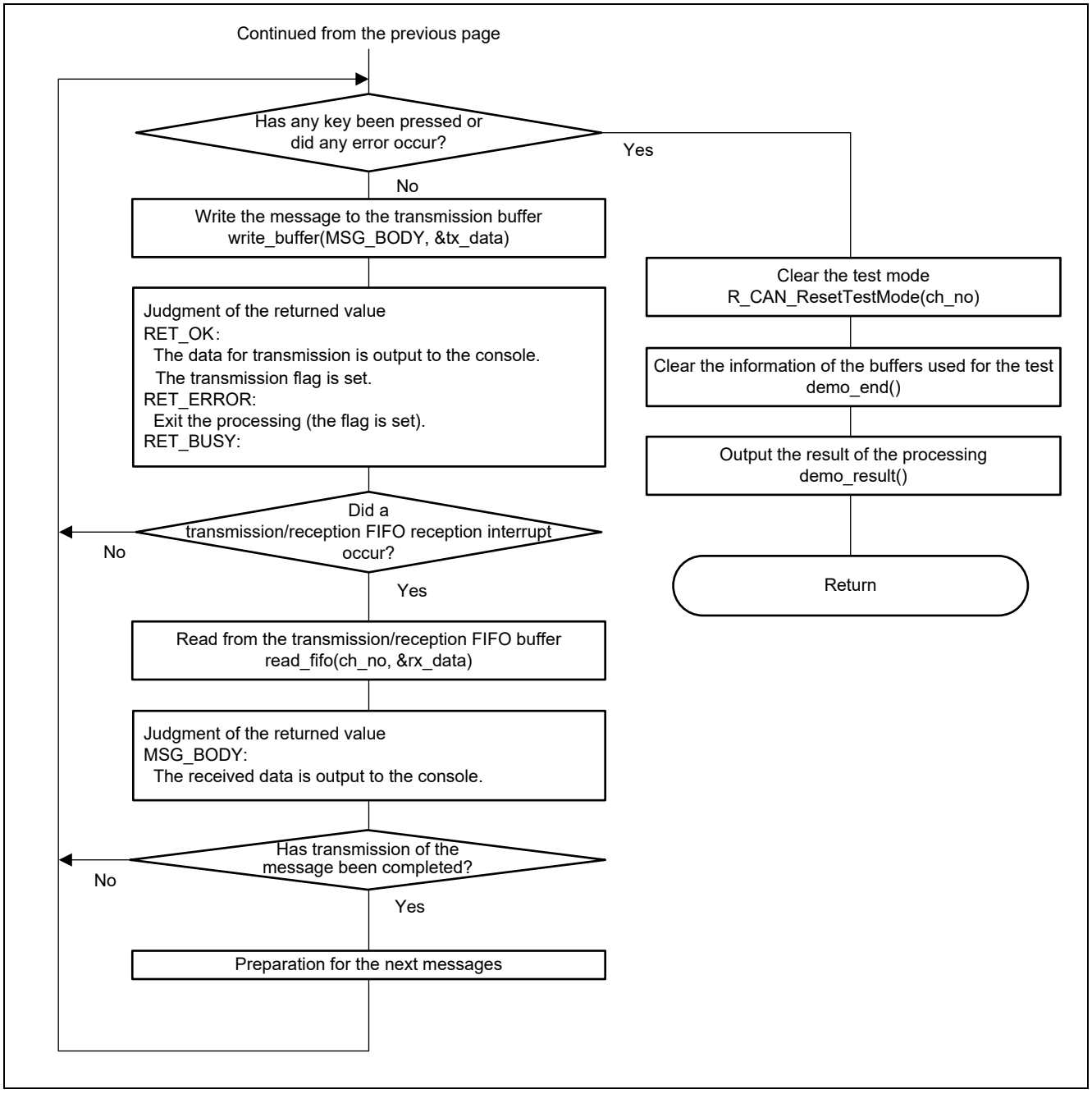


Figure 4-20 Test Which Sends a Message by Using the Transmission Buffer and Receives the Message by the Transmission/Reception FIFO Buffer (Reception Mode) in the Sample Code (2/2)

- Test which sends a message by using the transmission/reception FIFO buffer (transmission mode) and receives the message by the transmission/reception FIFO buffer (reception mode)

  Function name: void selftest_fifo_to_fifo(void)

  The figure below is a flowchart showing a test which sends a message by using the transmission/reception buffer (transmission mode) and receives the message by the transmission/reception FIFO buffer (reception mode).

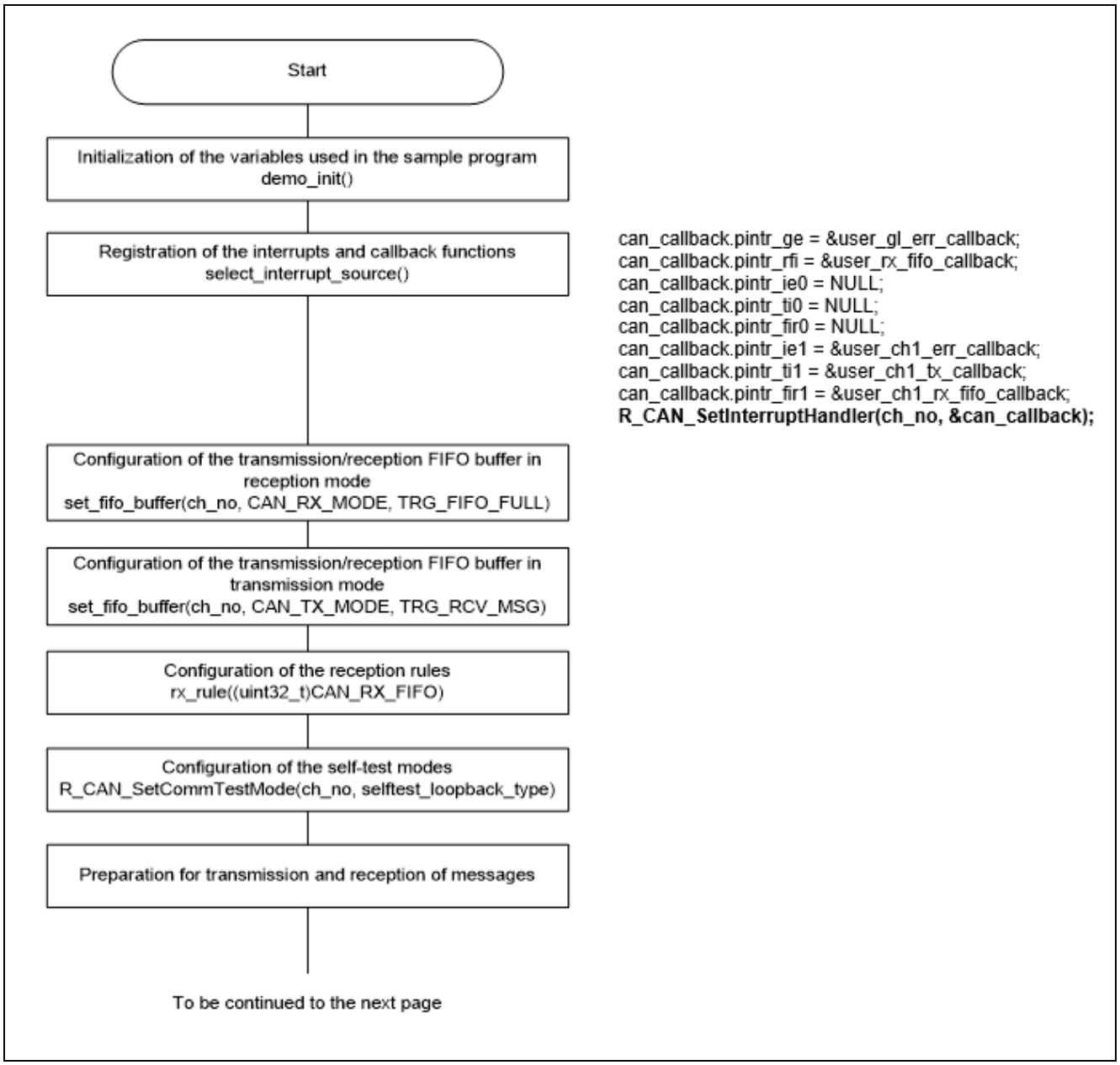


Figure 4-21 Test Which Sends a Message by Using the Transmission/Reception Buffer (Transmission Mode) and Receives the Message by the Transmission/Reception FIFO Buffer (Reception Mode) in the Sample Code (1/2)

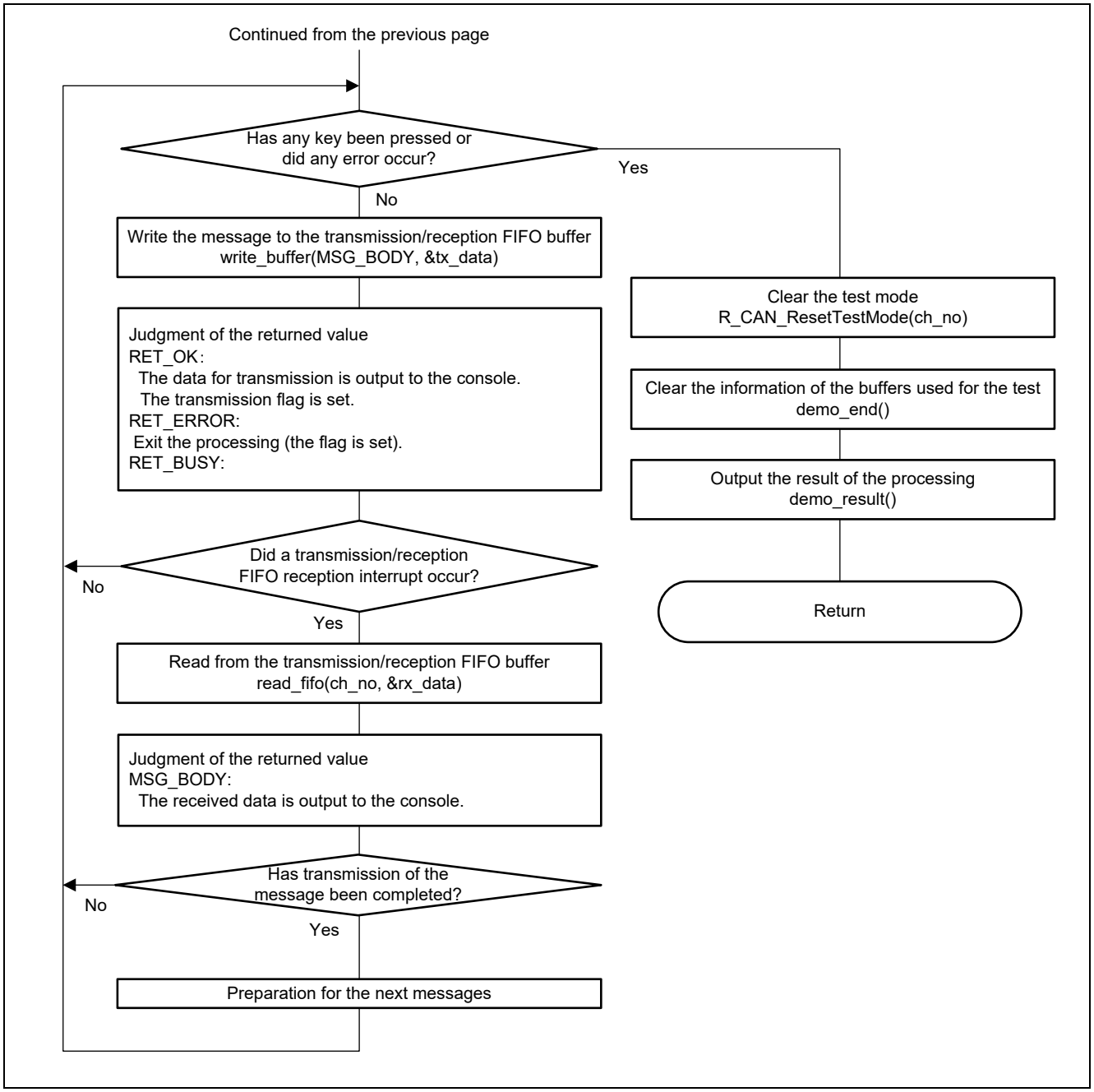


Figure 4-22 Test Which Sends a Message by Using the Transmission/Reception Buffer (Transmission Mode) and Receives the Message by the Transmission/Reception FIFO Buffer (Reception Mode) in the Sample Code (2/2)

4.5.6 Callback Processing

This sample program includes interrupt handling routines that are activated on occurrence of the various interrupt source conditions, and callback functions called by interrupt handling routines.

- RSCAN:CANGE (CAN global error)
  Interrupt handler: void user_gl_err_isr(void)
  Callback function: void user_gl_err_callback(void)

- RSCAN:CANIE1 (CAN1 error)
  Interrupt handler: void user_ch1_err_isr(void)
  Callback function: void user_ch1_err_callback(void)

- RSCAN:CANRFI (CAN reception FIFO interrupt)
  Interrupt handler: void user_rx_fifo_isr(void)
  Callback function: void user_rx_fifo_callback(void)

- RSCAN:CANTI1 (CAN1 transmission interrupt)
  Interrupt handler: void user_ch1_tx_isr(void)
  Callback function: void user_ch1_tx_callback(void)

- RSCAN:CANFIR1 (CAN1 transmission/reception FIFO reception completion interrupt)
  Interrupt handler: void user_ch1_rx_fifo_isr(void)
  Callback function: void user_ch1_rx_fifo_callback(void)

- SCIFA:RXIF2 (SCIFA reception FIFO data full interrupt)
  Interrupt handler: void scifa_key_input_isr(void)
  Callback function: void key_handler_callback(void)

- How to configure the interrupt handling routines is described here:

  Use the API functions below to configure the interrupt handling routines.

  void R_ICU_Regist(uint32_t vec_num, uint32_t type, uint32_t priority, uint32_t isr_addr);

  uint32_t vec_num: Vector number

  uint32_t type: Interrupt detection type

  uint32_t priority: Priority level of the interrupt

  uint32_t isr_addr: Address of the function for the interrupt handling routine

  Use the API functions below to enable or disable the interrupt.

  void R_ICU_Disable(uint32_t vec_num);

  void R_ICU_Enable(uint32_t vec_num);

  uint32_t vec_num: Vector number

- An example of configuring the callback function is described here:

  can_handle_t gb_can_handles[CAN_NUM];

```
typedef struct {
    void (*pintr_ge)(void);          /* pointer to user callback function. */
    void (*pintr_ie0)(void);         /* pointer to user callback function. */
    void (*pintr_ie1)(void);         /* pointer to user callback function. */
    void (*pintr_rfi)(void);         /* pointer to user callback function. */
    void
    (*pintr_fir0)(void);             /* pointer to user callback function. */
    void (*pintr_ti0)(void);         /* pointer to user callback function. */
    void
    (*pintr  fir1)(void);            /* pointer to user callback function. */

  typedef struct {
      bool                           ch_opened;
      can_callback_t                 can  callback;
```

Figure 4-23 to Figure 4-28 show the flowcharts for processing by the respective functions.

- (RSCAN:CANGE) – A callback function which is called on occurrence of a CAN global error

  Interrupt handler: void user_gl_err_isr(void)

  Callback function: void user_gl_err_callback(void)

  This callback function is called by the given interrupt handler on the occurrence of an error in the entire CAN module.

  The figure below is a flowchart showing the interrupt handling routine that is called when a global error occurs, and the callback function called in the process.

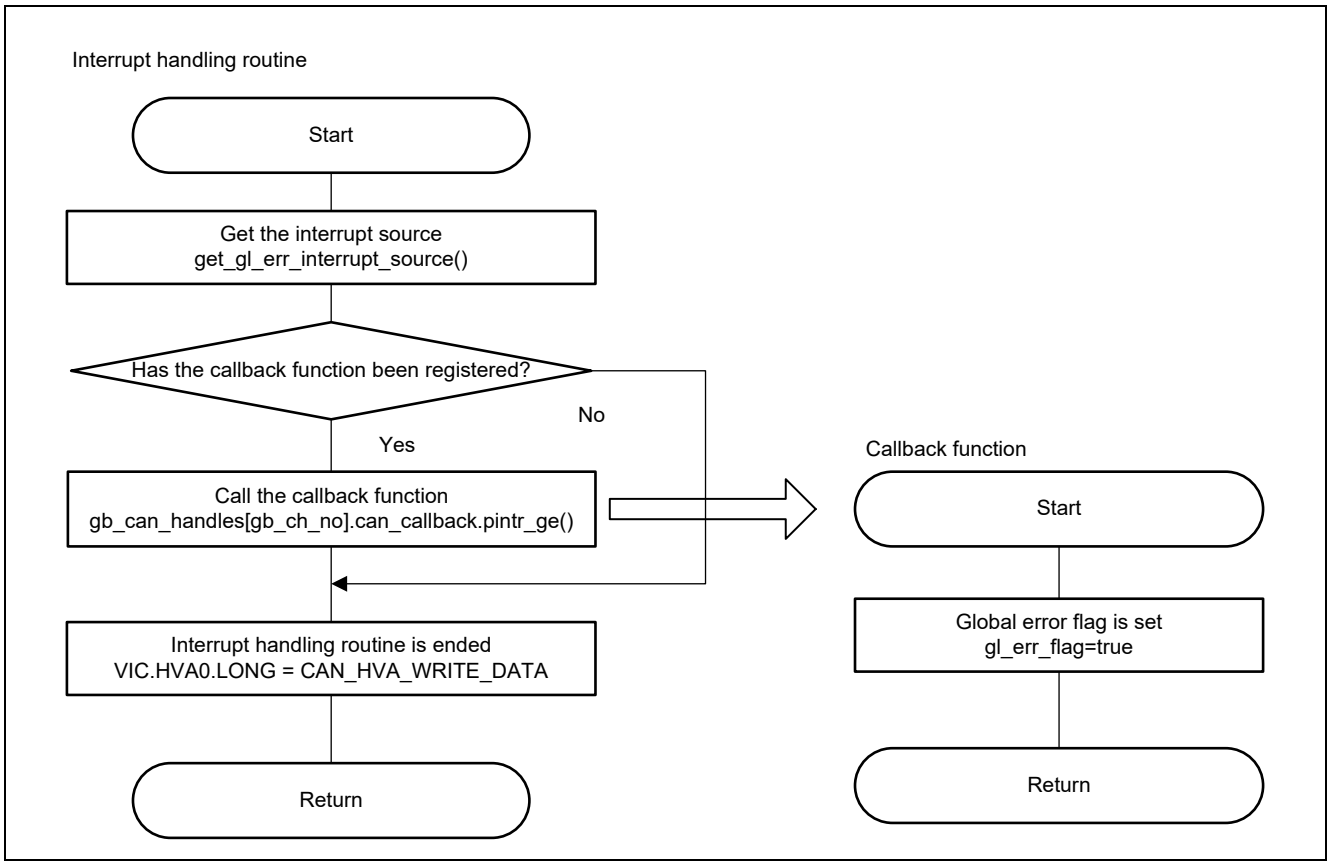


Figure 4-23 Handling of the Callback Function for a CAN Global Error in the Sample Code

- (RSCAN:CANIE1) – A callback function which is called on occurrence of a CAN1 error

  Interrupt handler: void user_ch1_err_isr(void)

  Callback function: void user_ch1_err_callback(void)

  This callback function is called by the given interrupt handler on the occurrence of an error in the CAN module channel (CAN1).

  The figure below is a flowchart showing the interrupt handling routine that is called when a channel error occurs, and the callback function called in the process.

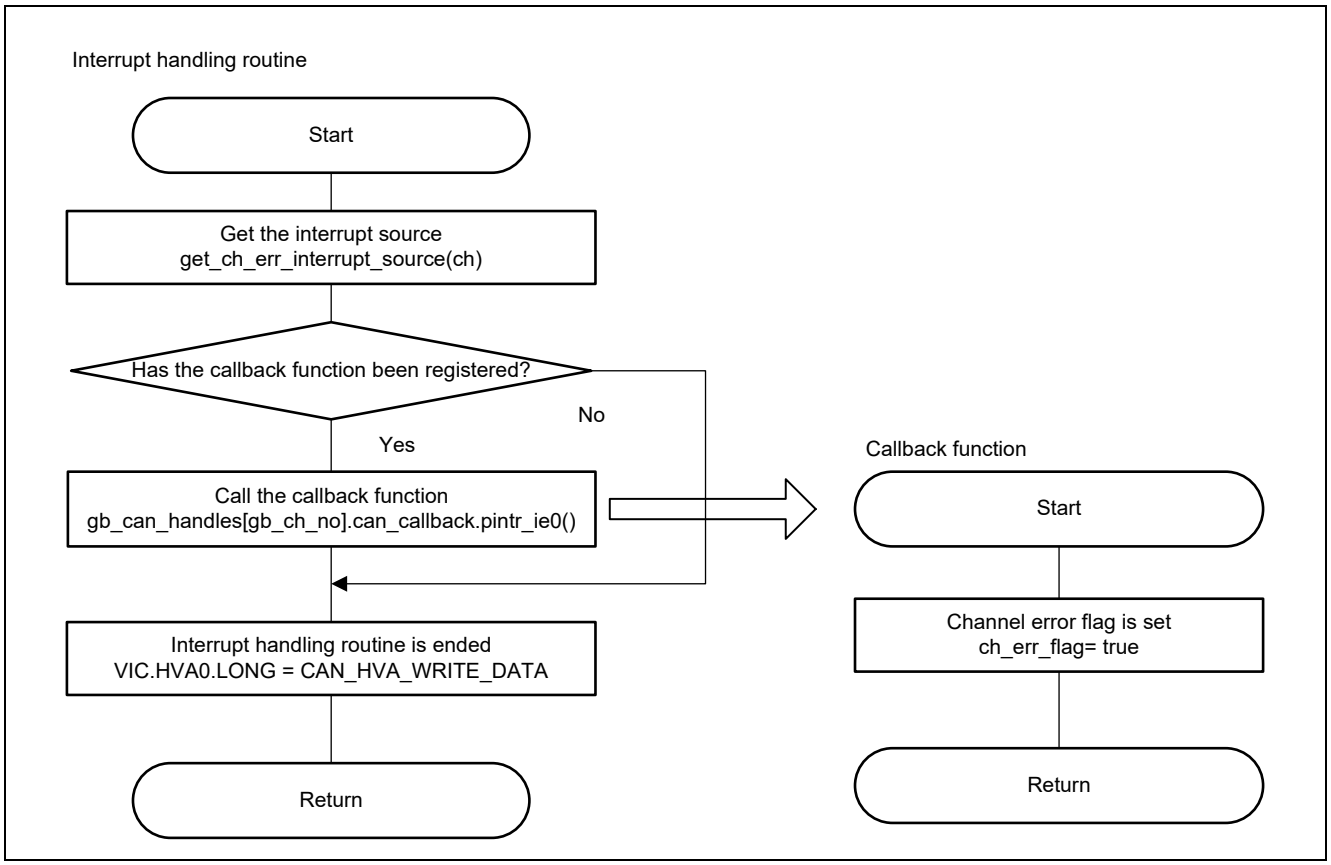


Figure 4-24 Handling of the Callback Function for a CAN1 Error in the Sample Code

- (RSCAN:CANRFI) – A callback function which is called on occurrence of a CAN reception FIFO interrupt

  Interrupt handler: void user_rx_fifo_isr(void)

  Callback function: void user_rx_fifo_callback(void)

  With the settings for the reception of messages by the reception FIFO buffer, the interrupt handler calls this callback function when the reception FIFO buffer becomes full of messages.

  The figure below is a flowchart showing the interrupt handling routine that is called when a reception FIFO buffer interrupt is generated, and the callback function called in the process.

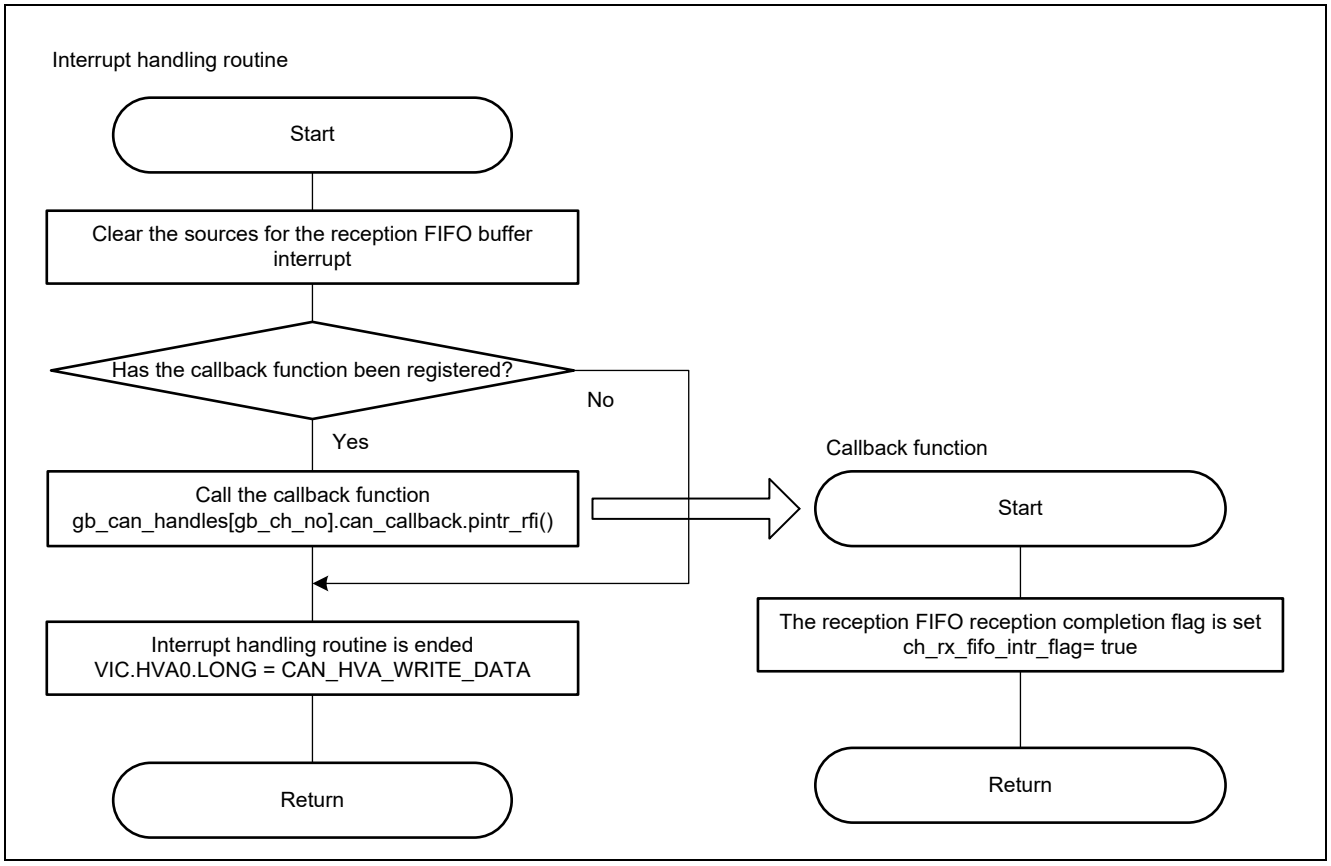


Figure 4-25 Handling of the Callback Function for a CAN Reception FIFO Interrupt in the Sample Code

- (RSCAN:CANTI1) – A callback function which is called on occurrence of a CAN1 transmission interrupt

  Interrupt handler: void user_ch1_tx_isr(void)

  Callback function: void user_ch1_tx_callback(void)

  This callback function is called by the given interrupt handler on completion of transmission of a message.

  The figure below is a flowchart showing the interrupt handling routine that is called when a transmission completion interrupt is generated, and the callback function called in the process.

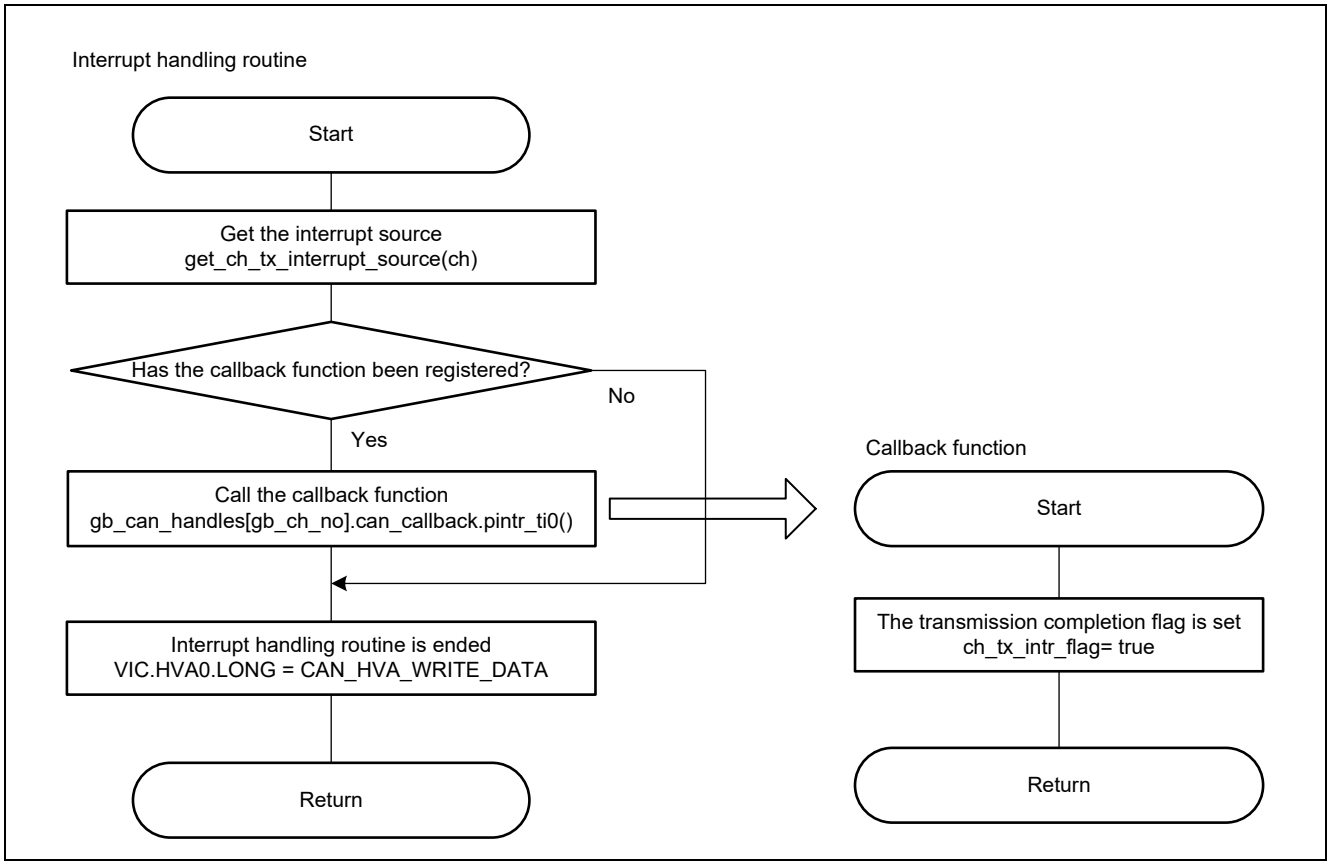


Figure 4-26 Handling of the Callback Function for a CAN1 Transmission Interrupt in the Sample Code

- (RSCAN:CANFIR1) – A callback function that is called on occurrence of a CAN1 transmission/reception FIFO reception interrupt

  Interrupt handler: void user_ch1_rx_fifo_isr(void)

  Callback function: void user_ch1_rx_fifo_callback(void)

  With the settings for the reception of messages by the transmission/reception FIFO buffer, the interrupt handler calls this callback function when the transmission/reception FIFO buffer becomes full of messages.

  The figure below is a flowchart showing the interrupt handling routine that is called when a transmission/reception FIFO reception completion interrupt is generated, and the callback function called in the process.

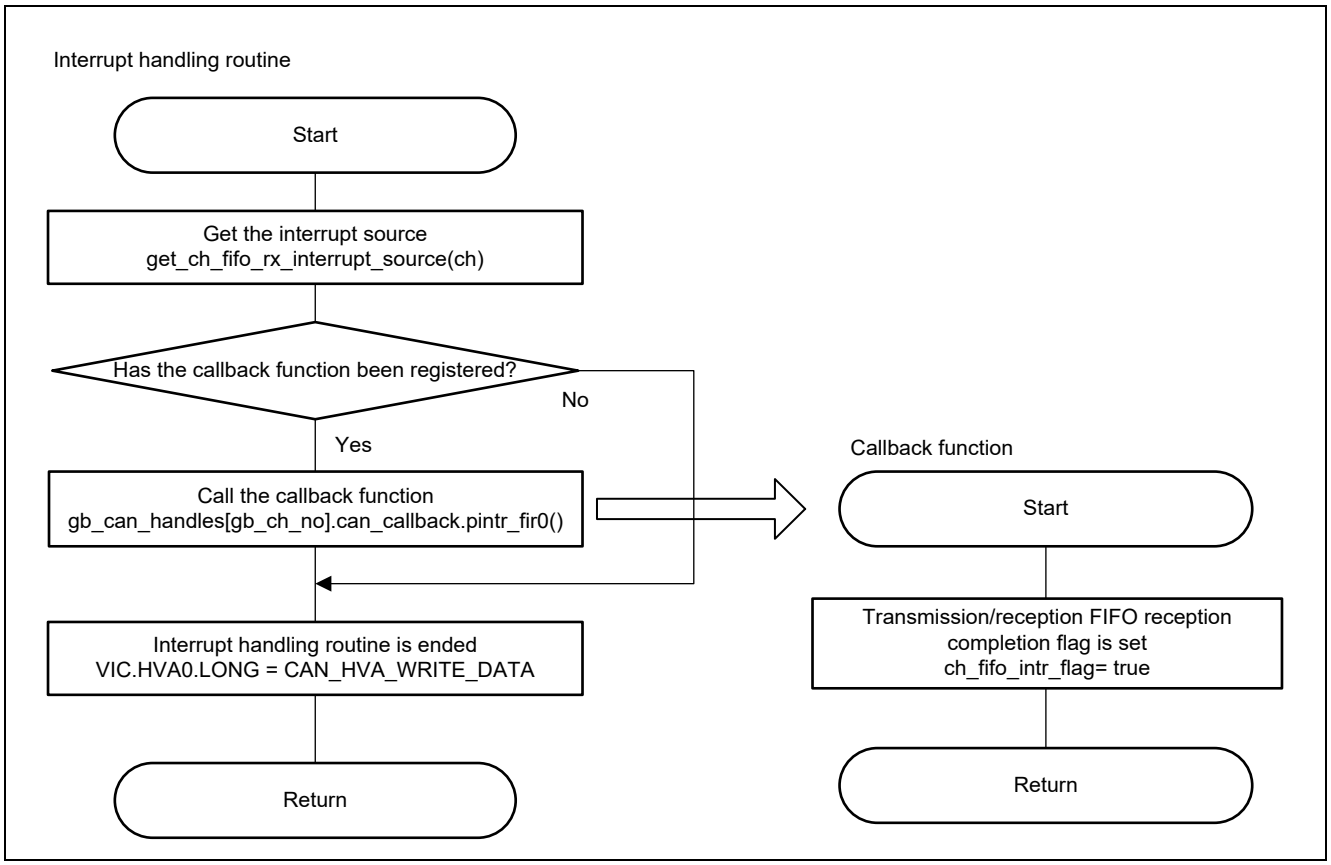


Figure 4-27 Handling of the Callback Function for a CAN1 Transmission/Reception FIFO Reception Completion Interrupt in the Sample Code

- (SCIFA:RXIF2) – A callback function which is called on occurrence of a reception FIFO data full interrupt

  Interrupt handler: void scifa_key_input_isr(void)

  Callback function: void key_handler_callback(void)

  This callback function is called from the handling routine for the pressing of a key producing an interrupt from the SCIFA module.

  When this sample program is being used for transmission, messages are repeatedly transmitted to the receiving side.

  The following callback function is provided to stop the ongoing transmission of a message.

  The transmission is stopped by pressing any key, which causes the SCIFA module to set a flag.

  The figure below is a flowchart showing the interrupt handling routine that is called when a key input interrupt is generated, and the callback function called in the process.

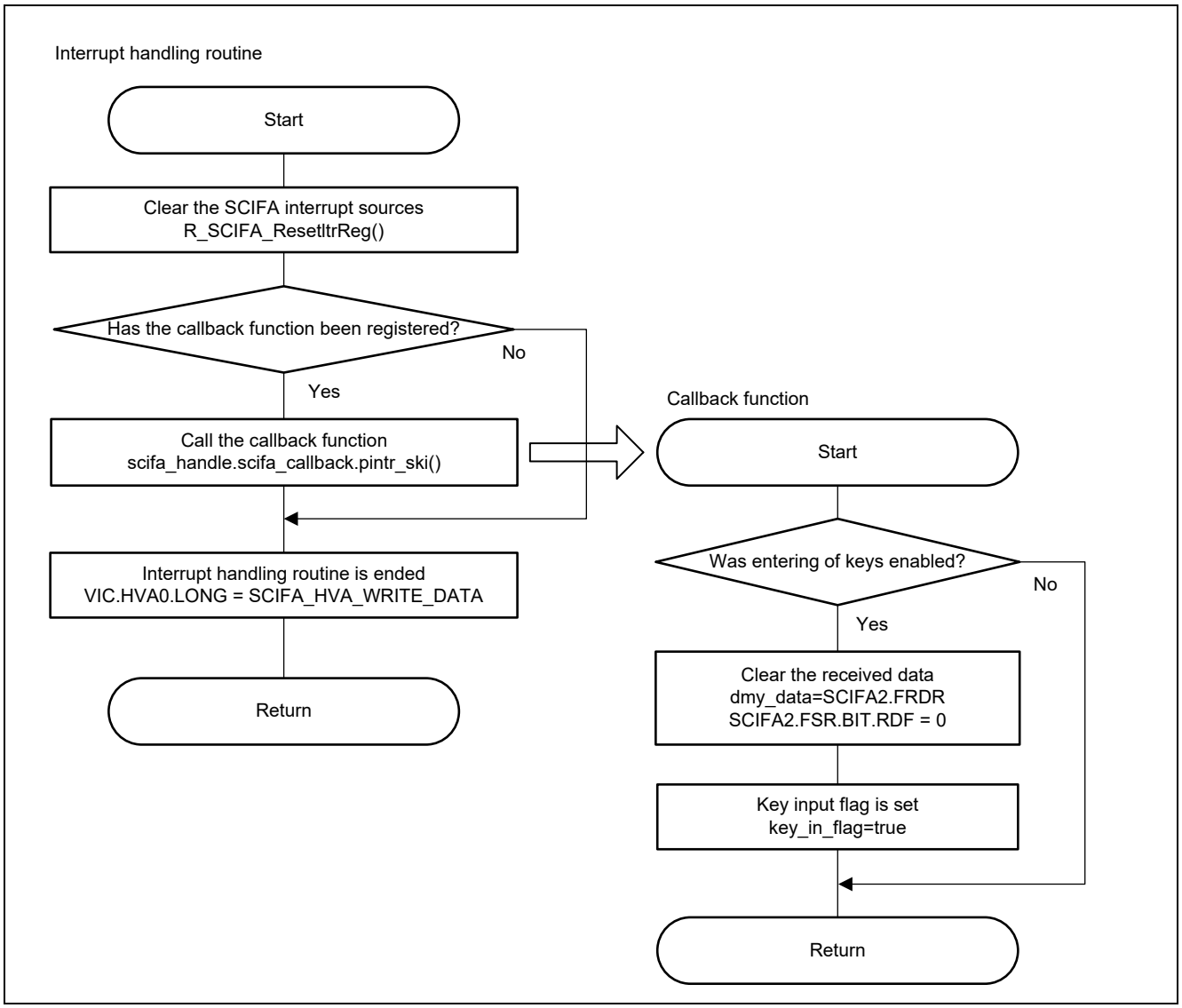


Figure 4-28 Handling of the Callback Function for a Reception FIFO Data Full Interrupt of SCIFA in the Sample Code

4.6 Tutorials

4.6.1 Operational Overview

Table 4-4 summarizes the functions of the RSCAN sample program. Figure 4-29 shows a system block diagram.

Table 4-4 Operational Overview

Function	Description
Multiplexed pin settings	- PC7: CAN1 CRXD1 - P66: CAN1 CTXD1
CAN communication	- Channel 1 (CAN1) is used.
Interrupt sources (interrupt priority)	- CAN global error (5) - CAN1 error (5) - CAN reception FIFO (5) - CAN1 transmission/reception FIFO reception completion (5) - CAN1 transmission (5)
Transfer rate setting	- 1 Mbps
Operating modes	- Transmission mode (transmitting side when two evaluation boards are connected) Message transmission by using the transmission buffer Message transmission by using the transmission/reception FIFO buffer in transmission mode - Reception mode (receiving side when two evaluation boards are connected) Message reception by using the reception buffer Message reception by using the reception FIFO buffer Message reception by using a transmission/reception FIFO buffer (reception mode) - Test for transmission while receiving data at the same time - Test modes (self tests only with a single evaluation board) Self-test mode 0 (external loopback mode) Self-test mode 1 (internal loopback mode)
Operation	Operating mode is selected from the menu.
Operation result display	The result is output to the console.

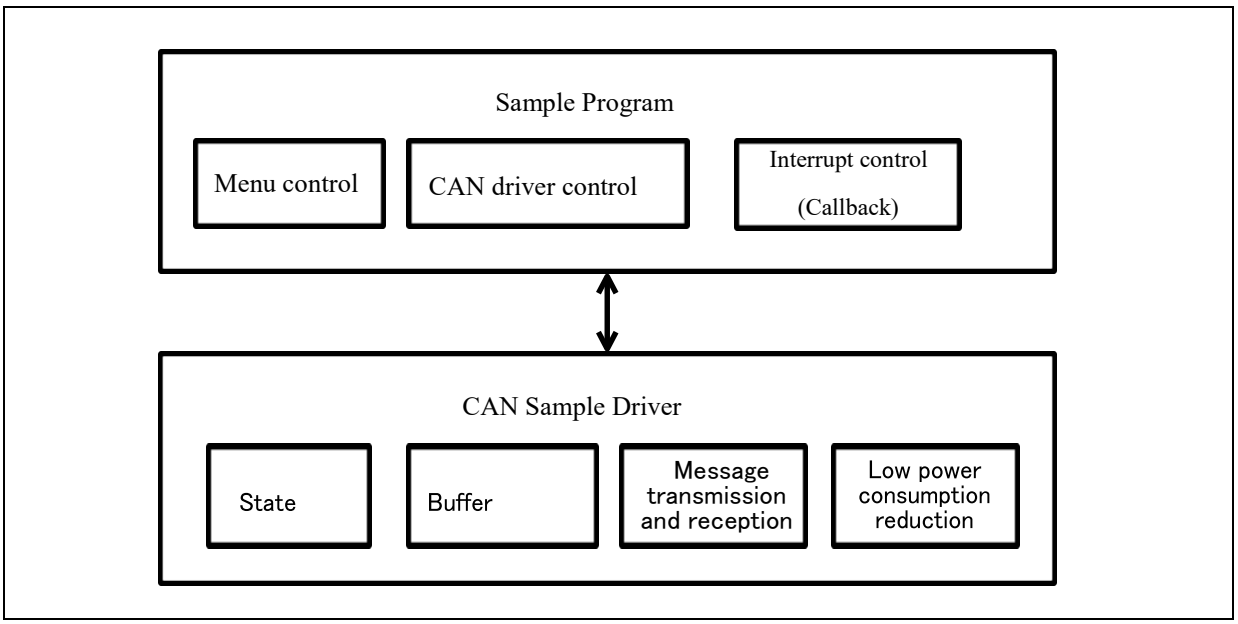


Figure 4-29 System Block Diagram

4.6.2 Preparations (Self Tests)

Self tests of CAN communications can be performed by using this sample program. Testing CAN communications is possible by using only the evaluation board which is connected to the development environment.

4.6.3 Preparations (Transmission and Reception Test)

A transmission and reception test requires two evaluation boards: evaluation board A which is connected to the development environment and evaluation board B which is connected to the development environment that is set up in another computer.

Connect evaluation boards A and B with a CAN cable[*1]. (Connect to the CAN1 connector of each board.)

Note 1.

Connect the CAN-H pin of the CAN connector 1 (J4) of board A to the CAN-H pin of the CAN connector 1 (J4) of board B.

Connect the CAN-L pin of the CAN connector 1 (J4) of board A to the CAN-L pin of the CAN connector 1 (J4) of board B.

4.6.4 Terminal Software (Tera Term)

This sample program performs RS-232C COM port communications with the host computer by using asynchronous communication of the serial communications interface with FIFO (SCIFA).

Start the terminal software (Tera Term) at the host computer and make serial port settings (baud rate: 115200). Set the line feed code during transmission and reception to CR.

- Transfer rate: 115,200 bps
- Character length: 8 bits
- Stop bit length: 1 bit
- Parity function: None
- Hardware flow control: None

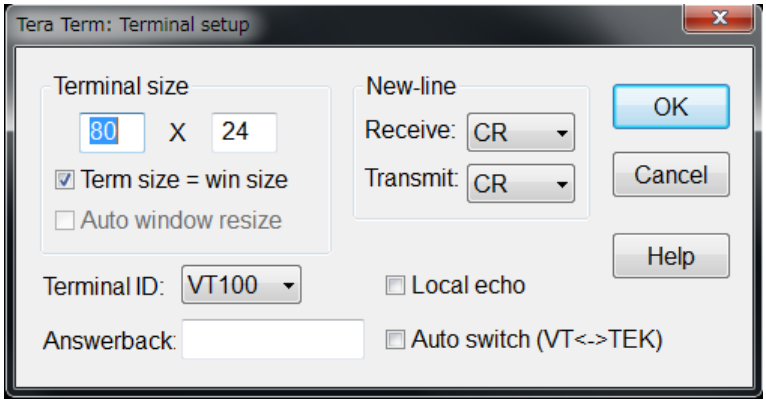


Figure 4-30 Tera Term: Terminal Settings

The screenshot below shows how to set the serial port to COM4.

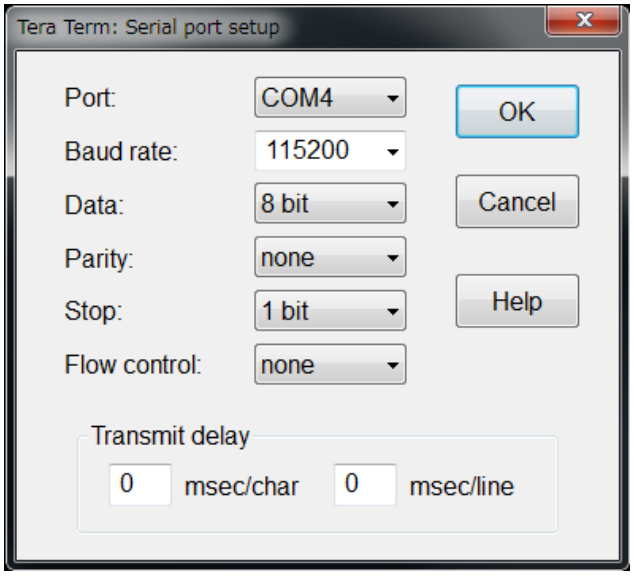


Figure 4-31 Tera Term: Serial Port Settings

4.6.5 Functions of the Sample Program

Start the terminal software (Tera Term), and then start this sample program.

Select a function of the sample program from the console menu.

- Main menu

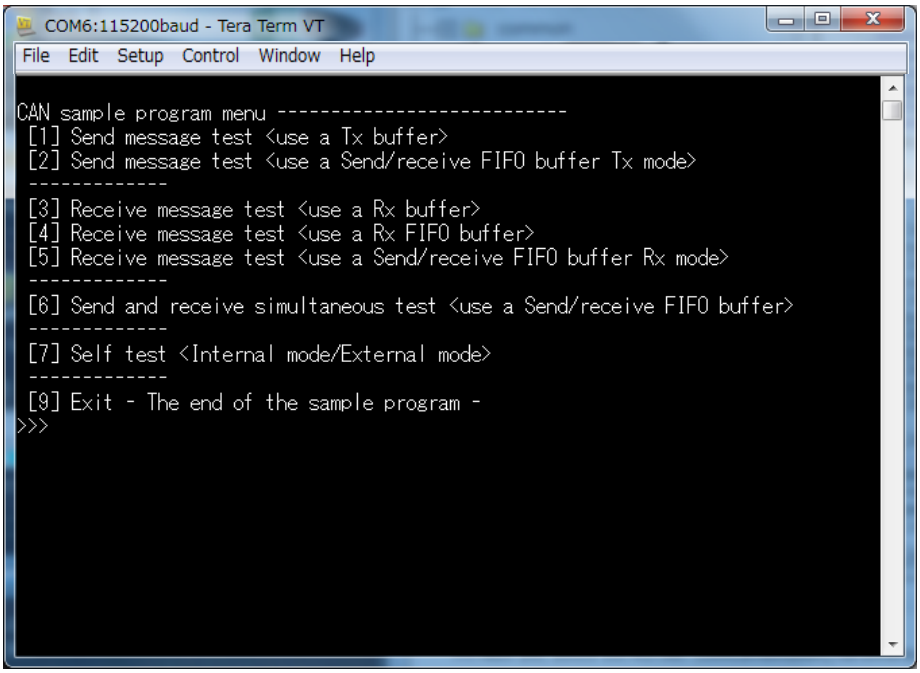


Figure 4-32 Main Menu of the Sample Program

The menu provides the following functions:

- [1] Send message test <uses a Tx buffer>

  Test that sends a message from the transmission buffer
- [2] Send message test <uses a Send/receive FIFO buffer Tx mode>

  Test that sends a message from the transmission/reception FIFO buffer (transmission mode)
- [3] Receive message test <uses a Rx buffer>

  Test that receives a message by the reception buffer
- [4] Receive message test <uses a Rx FIFO buffer>

  Test that receives a message by the reception FIFO buffer
- [5] Receive message test <uses a Send/receive FIFO buffer Rx mode>

  Test that receives a message by the transmission/reception FIFO buffer (reception buffer)
- [6] Send and receive simultaneous test < uses a Send/receive FIFO buffer >

  Test for transmission while receiving data at the same time
- [7] Self test <Internal mode/External mode>

  Self-test selection menu
- [9] Exit – The end of the sample program –

  Exit from the sample program

- Self-test menu

  This test enables a self test only with evaluation board A (external loopback and internal loopback).

  The external loopback mode and the internal loopback mode can be switched from the menu.

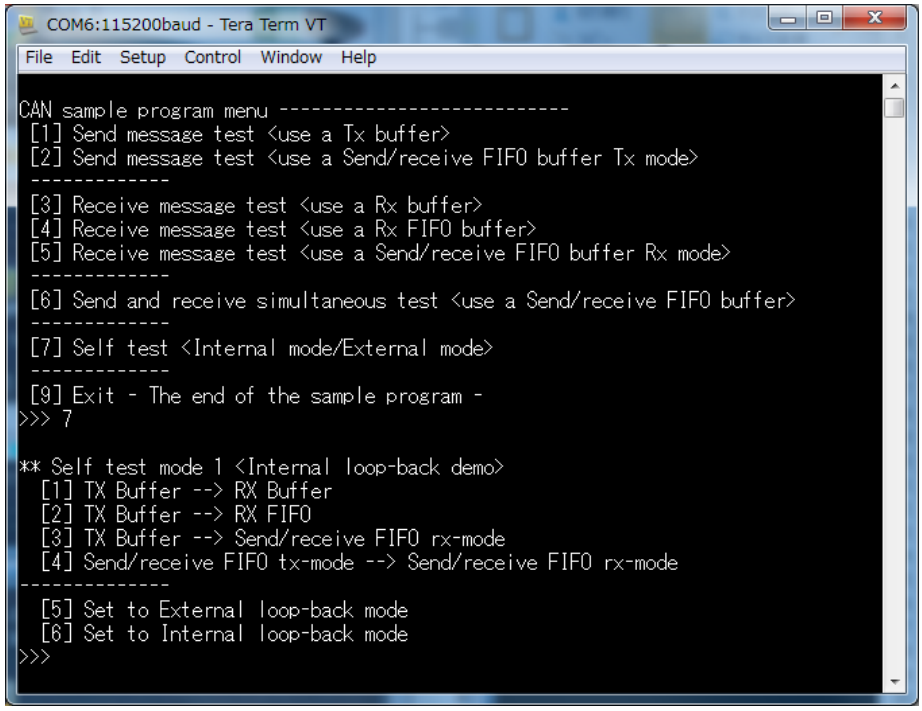


Figure 4-33 Self Test Selection Menu when [7] Self test <Internal/External> Is Selected

The self test menu provides the following functions:

- 1: Tx Buffer → Rx Buffer

  Test which sends a message from the transmission buffer and receives the message by the reception buffer

- 2: Tx Buffer → Rx FIFO buffer

  Test which sends a message from the transmission buffer and receives the message by the reception FIFO buffer

- 3: Tx Buffer → Send/receive FIFO buffer Rx mode

  Test which sends a message from the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode)

- 4: Send/receive FIFO buffer Tx mode → Send/receive FIFO buffer Rx mode

  Test which sends a message from the transmission/reception FIFO buffer (transmission mode) and receives the message by the transmission/reception FIFO buffer (reception mode)

- 5: Set to External loop back mode

  Selection of self-test mode 0 (external loopback mode)

- 6: Set to Internal loop back mode

  Selection of self-test mode 1 (internal loopback mode)

4.6.6 Sample Program Settings

This sample program operates with the following settings:

- Channel: CAN1 (fixed)
- Transfer rate: 1 Mbps (fixed)
- Message for transmission: Repeated transmission of one message (8 bytes)
- Reception rule: 1 rule (Only the message with message ID 0x120 can be received) (fixed)
- Transmission buffer number: 0 (fixed)
- Reception buffer number: 1 (fixed)
- Transmission/reception FIFO buffer number: 0 (fixed)
- Transmission/reception FIFO buffer number (transmission mode): 0 (fixed)
- Transmission/reception FIFO buffer number (reception mode): 1 (fixed)
- Transmission buffer number to be linked to the transmission/reception FIFO buffer: 2 (fixed)

Sample data

- Message ID: 0x120 (fixed)
- Message type: Standard ID (fixed)
- Data format: Data frame (fixed)
- Message data: 00h to FFh (8 bytes are sent as one message)

Reception buffer

- Buffer size: 1,024 bytes (if data exceeding this size is received, the buffer is overwritten from the beginning)

4.6.7 Transmission Test

- Preparations

  By using a cable, connect evaluation board A which is connected to the development environment to evaluation board B which is connected to the development environment that is set up in another computer. See section 4.6.3, Preparations (Transmission and Reception Test).

- Procedure

  First, place the receiving side (evaluation board B which is connected to the development environment that is set up in another computer) in reception mode.

  (Select [3], [4] or [5] from the main menu on the evaluation board B side.)

  Next, select [1] or [2] in the main menu on the transmitting side (evaluation board A).

  Repeat transmission of a message from the transmitting side (evaluation board A).

  Press any key on the transmitting side to stop the transmission test.

  The transmission test will end. The receiving side will also exit the reception mode.

- Data for transmission

  Message ID: 0x120

  Message type: Standard ID (value: 0)

  Data format: Data frame (value: 0)

  Message data: A message for transmission consists of eight bytes.

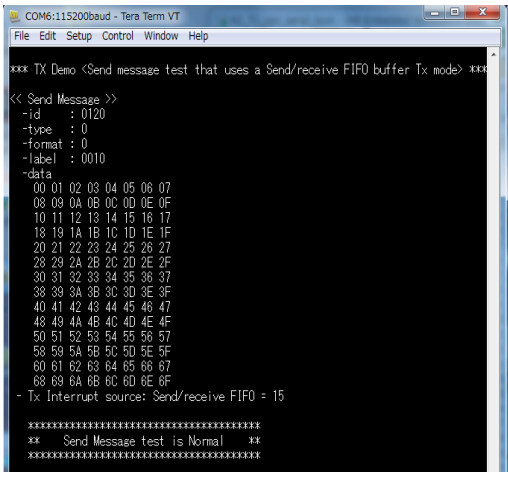
  (The sample uses continuous data from 0x00 to 0xFF.)

  Delimiter code: Message end code

  (The sample identifies eight continuous bytes of 0x00 as the end of the message.)

- Results
  The screenshots below show the results of message transmission test using the transmission/reception FIFO buffer (transmission mode).

Log output on the transmitting side



Log output on the receiving side

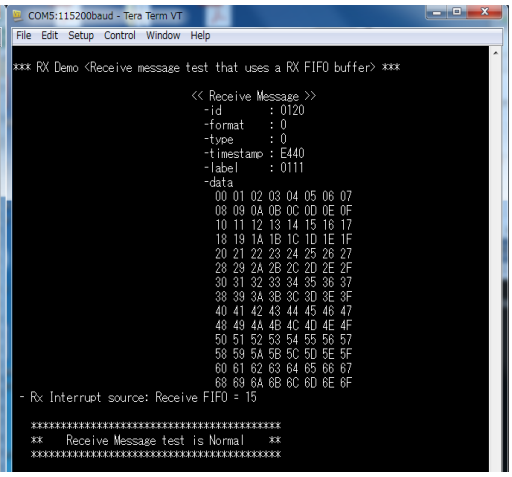


Figure 4-34 Example of Transmission Test Results by the Sample Program

- Remarks

  In the procedure above, if the message is sent from the transmitting side (evaluation board A) when the receiving side (evaluation board B that is connected to the development environment that is set up in another computer) is not in reception mode yet, the following error occurs.

  Send the message from the transmitting side (evaluation board A) after the receiving side (evaluation board B) gets ready.

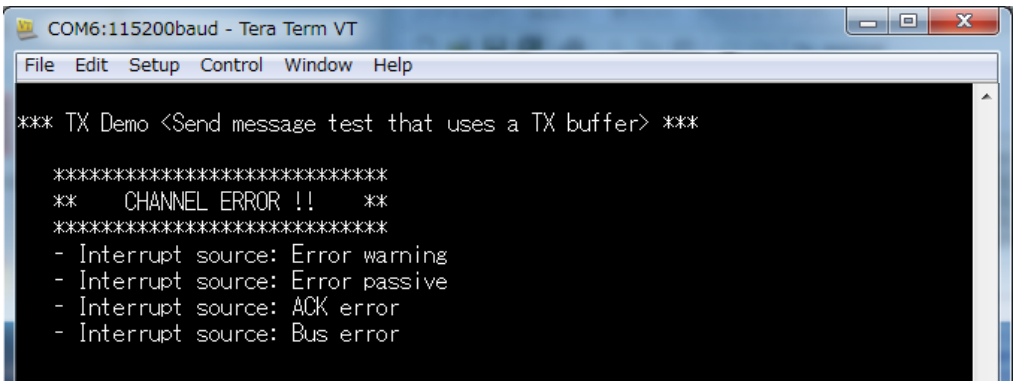


Figure 4-35 Example of Transmission Test Error Results by the Sample Program

4.6.8 Reception Test

- Preparations

  By using a cable, connect evaluation board A which is connected to the development environment to evaluation board B which is connected to the development environment that is set up in another computer. See section 4.6.3, Preparations (Transmission and Reception Test).

- Procedure

  First, select [3], [4] or [5] from the main menu on the evaluation board A side to place it in reception mode.

  Next select [1] or [2] in the main menu on the evaluation board B side that is connected to the development environment that is set up in another computer.

  Repeat transmission of a message from the transmitting side (evaluation board B).

  Press any key on the transmitting side (evaluation board B) to stop the reception test on the receiving side (evaluation board A).

  The transmitting side (evaluation board B) will end the transmission test and the receiving side (evaluation board A) will exit from reception mode.

- Results

  The screenshots below show the results of message reception test using the transmission/reception FIFO buffer (reception mode).

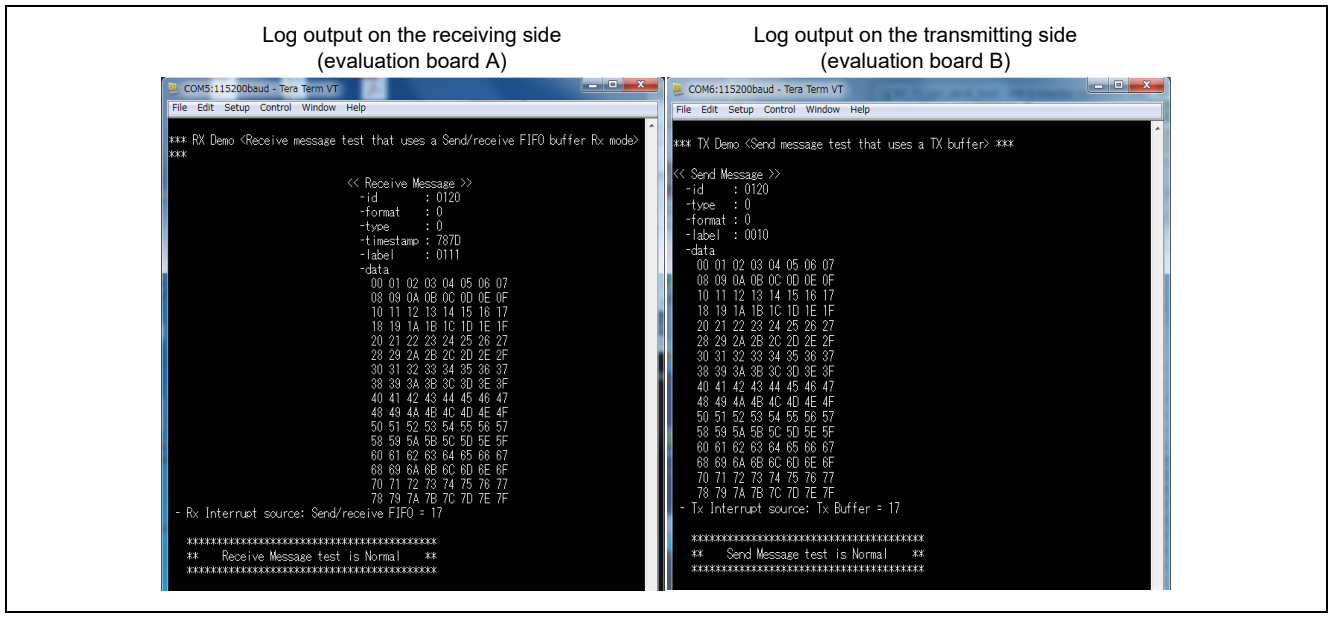


Figure 4-36 Example of Reception Test Results by the Sample Program

4.6.9 Test for Transmission While Receiving Data at the Same Time

- Preparations

  By using a cable, connect evaluation board A which is connected to the development environment to evaluation board B which is connected to the development environment that is set up in another computer. See section 4.6.3, Preparations (Transmission and Reception Test).

- Procedure

  1. Select [6] in the main menu on the evaluation board A side.

  2. A guide that prompts you to press a key will be displayed.

  3. Select [6] similarly in the main menu on the evaluation board B side that is connected to the development environment that is set up in another computer.

  4. A guide that prompts you to press a key will be displayed.

  5. After confirming that a guide that prompts you to press a key is displayed on both sides, press any key on the evaluation board B side.

  By performing steps 1 to 5, a message is sent repeatedly from both sides of evaluation boards A and B.

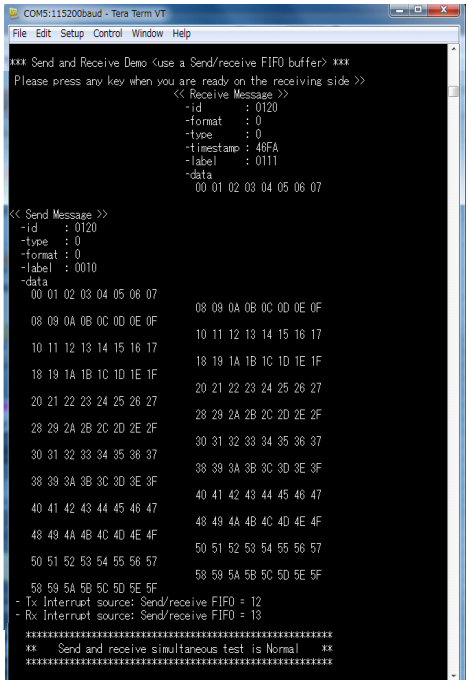
  Press any key on the evaluation A or B side to stop the test.

  The test for transmission while receiving data at the same time will end on both sides of evaluation boards A and B.

- Results

  The screenshots below show the results of the test for transmission while receiving data at the same time.

Log output on the evaluation board A
Transmitting messages to B
while receiving messages from B



Log output on the evaluation board B
Receiving messages from A
while transmitting messages to A

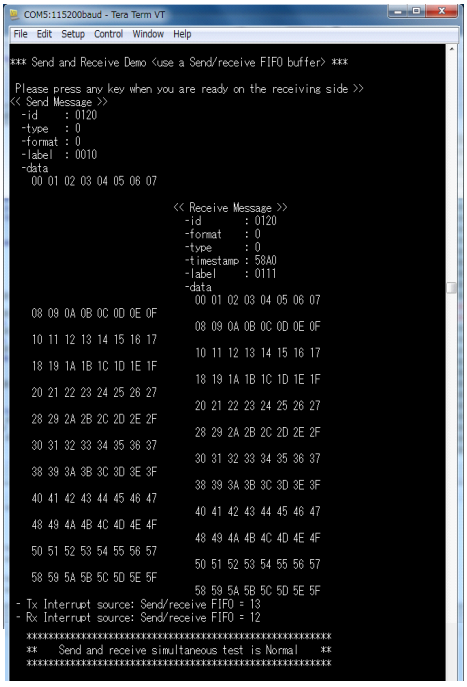


Figure 4-37 Example of Test Results for Transmission While Receiving Data at the Same Time by the Sample Program

4.6.10 **Self Tests**

- Preparations

  Only evaluation board A that is connected to the development environment is used. See section 4.6.2, Preparations (Self Tests).

- Procedure

  Select [7] in the main menu on the evaluation board A side to display the self test menu.

  Select a desired test item from the self test menu.

  Self tests are available in external loopback mode and in internal loopback mode.

  Select [5] or [6] in the self test menu. (The default is the internal loopback mode.)

- Results

  The screenshot below shows the result of a self test which sends a message from the transmission buffer and receives the message by the transmission/reception FIFO buffer (reception mode).

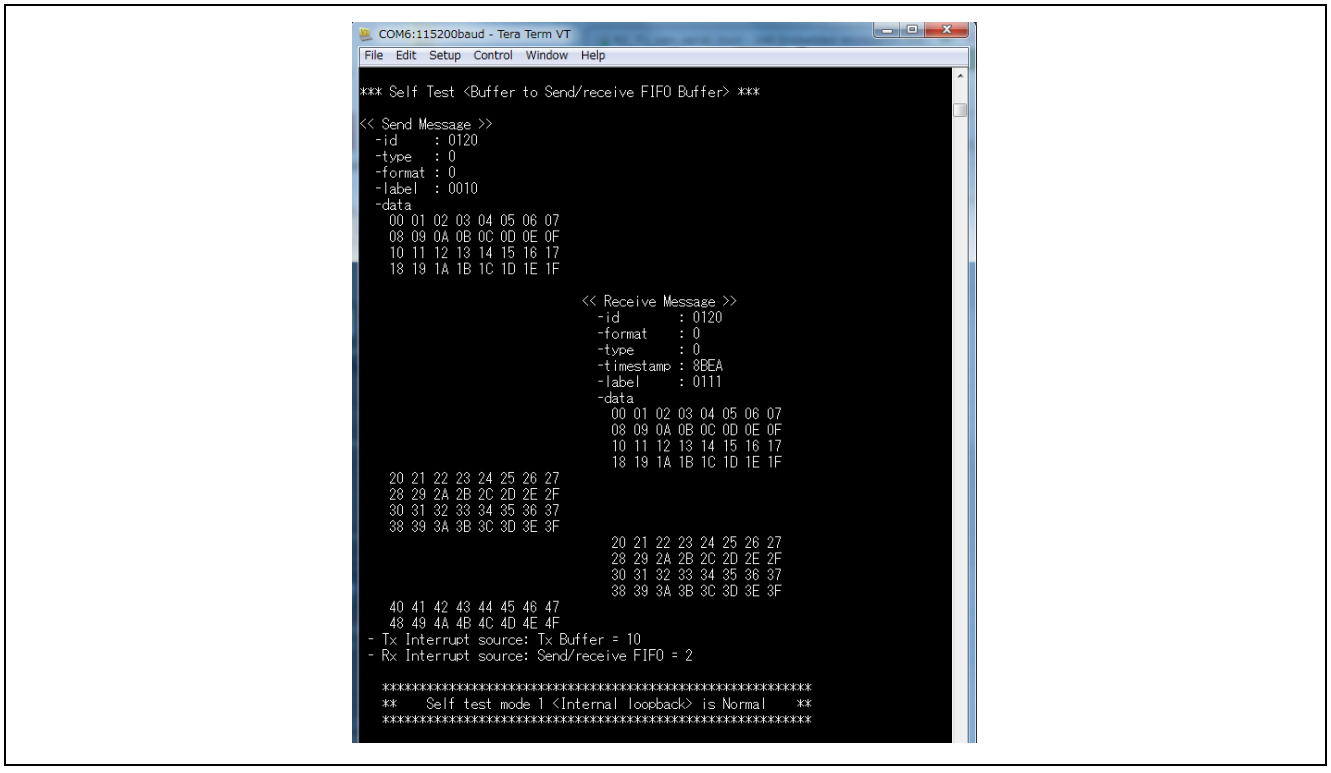


Figure 4-38 Example of Self Test Results by the Sample Program

5. RSPI Sample Software

5.1 Overview

This section describes the sample program that uses the RSPI driver that controls the RSPI controller 0 channel (RSPI0) installed on the evaluation board.

The features of the RSPI sample program are as follows.

By connecting a computer and by using terminal software, the RSPI communication data transmission and reception results between the master and the slave can be confirmed on the computer screen.

Target Devices

EC-1

When applying the sample program covered in this application note to another microcontroller, modify the program according to the specifications for the target microcontroller and conduct an extensive evaluation of the modified program.

5.2 Functions

A list of functions is shown in Table 5-1 List of Functions.

Table 5-1 List of Functions

Function Name	Description	Scope	Definition File
main	Main processing	global	main.c
cmt_standby	Stops the CMT module	local	main.c
get_sw1	Gets the switch 1 status	local	main.c
get_sw8	Gets the switch 8 status	local	main.c
board_init	Initializes the board settings	global	board_RenesaEva.c
board_output	Stores the board output data	global	board_RenesaEva.c
board_input	Gets the input port status	global	board_RenesaEva.c
board_input_dipsw	Gets the DIP switch status	global	board_RenesaEva.c
board_rspi_init	Initializes the RSPI port	global	board_RenesaEva.c

5.3 Details of the Functions

5.3.1 main

(1) **Synopsis**

Main processing of the sample program

(2) **C language format**

int main (void);

(3) **Parameters**

None

(4) **Description**

This function is the main processing of sample program.
For details on the processing, see section 5.4.1, Main Processing.

(5) **Returned values**

None

5.3.2 cmt_standby

(1) **Synopsis**

Stops the CMT module.

(2) **C language format**

static void cmt_standby (void);

(3) **Parameters**

None

(4) **Description**

This function stops the CMT module.
It disables writing to the low power consumption function and the registers related to resets.

(5) **Returned values**

None

5.3.3 get_sw1

(1) **Synopsis**

Gets the switch 1 status.

(2) **C language format**

static uint8_t get_sw1(void);

(3) **Parameters**

None

(4) **Description**

This function gets the status of switch 1.

(5) **Returned values**

Returned Value	Meaning
tmp_port	Status of the switch

5.3.4 get_sw8

(1) **Synopsis**

Gets the switch 8 status.

(2) **C language format**

static uint8_t get_sw8(void);

(3) **Parameters**

None

(4) **Description**

This function gets the status of switch 8.

(5) **Returned values**

Returned Value	Meaning
tmp_port	Status of the switch

5.3.5 board_init

(1) Synopsis

Initializes the board settings.

(2) C language format

void board_init(void);

(3) Parameters

None

(4) Description

This function initializes the board settings.

(5) Returned values

None

5.3.6 board_output

(1) Synopsis

Stores the board output data.

(2) C language format

void board_output(uint8_t output);

(3) Parameters

I/O	Parameter		Description
I	uint8_t	output	Output data value

(4) Description

This function writes the output data to the port output data register.

(5) Returned values

None

5.3.7 board_input

(1) Synopsis

Gets the input port status.

(2) C language format

uint8_t board_input(void);

(3) Parameters

None

(4) Description

This function gets the status of the input port pin.

(5) Returned values

Returned Value	Meaning
uint8_t	Status of the input port

5.3.8 board_input_dipsw

(1) Synopsis

Gets the DIP switch status.

(2) C language format

uint8_t board_input_dipsw (void);

(3) Parameters

None

(4) Description

This function gets the status of the DIP switch.

(5) Returned values

Returned Value	Meaning
uint8_t	Status of the DIP switch

5.3.9 board_rspi_init

(1) **Synopsis**

Initializes the RSPI port.

(2) **C language format**

void board_rspi_init(void);

(3) **Parameters**

None

(4) **Description**

This function initializes the RSPI port settings.

(5) **Returned values**

None

5.4 Flowcharts

5.4.1 Main Processing

The figure below is a flowchart of the main processing of the sample code.

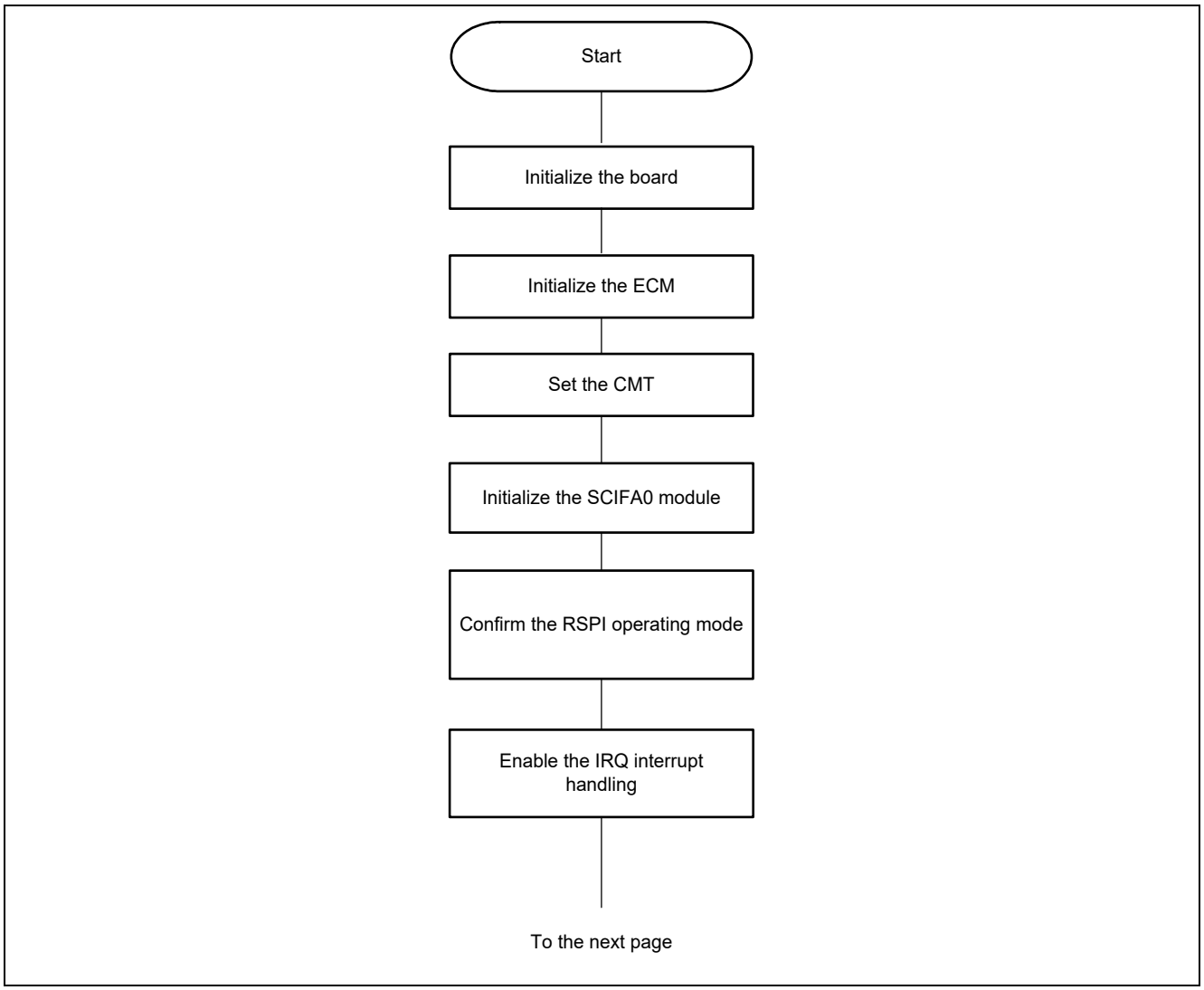


Figure 5-1 Main Processing of the Sample Code (1/2)

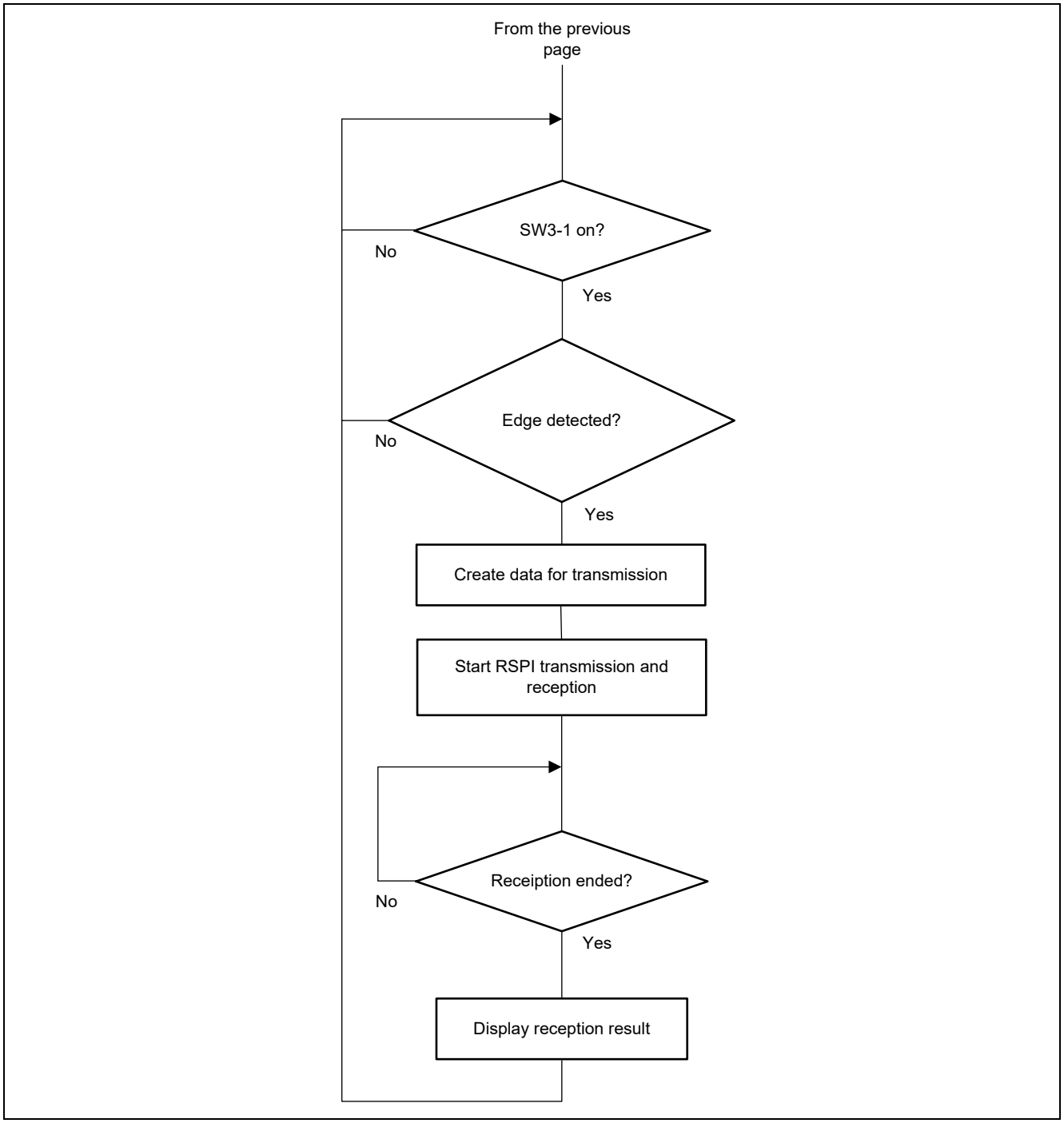


Figure 5-2 Main Processing of the Sample Code (2/2)

5.5 Tutorials

5.5.1 Operational Overview

Table 5-2 summarizes the functions of the RSPI sample program. Figure 5-3 shows a system block diagram.

Table 5-2 Operational Overview

Function	Description
Communication channel	Channel 0 is used.
Operating modes	- SPI operation: 3-wire - Communication mode: Master mode (communication only) - Data length: 16 bits - Bit rate: 6.25 Mbps in slave mode
Operation	The operating mode is switched by the DIP switch.
Operation result display	The result is output to the console.

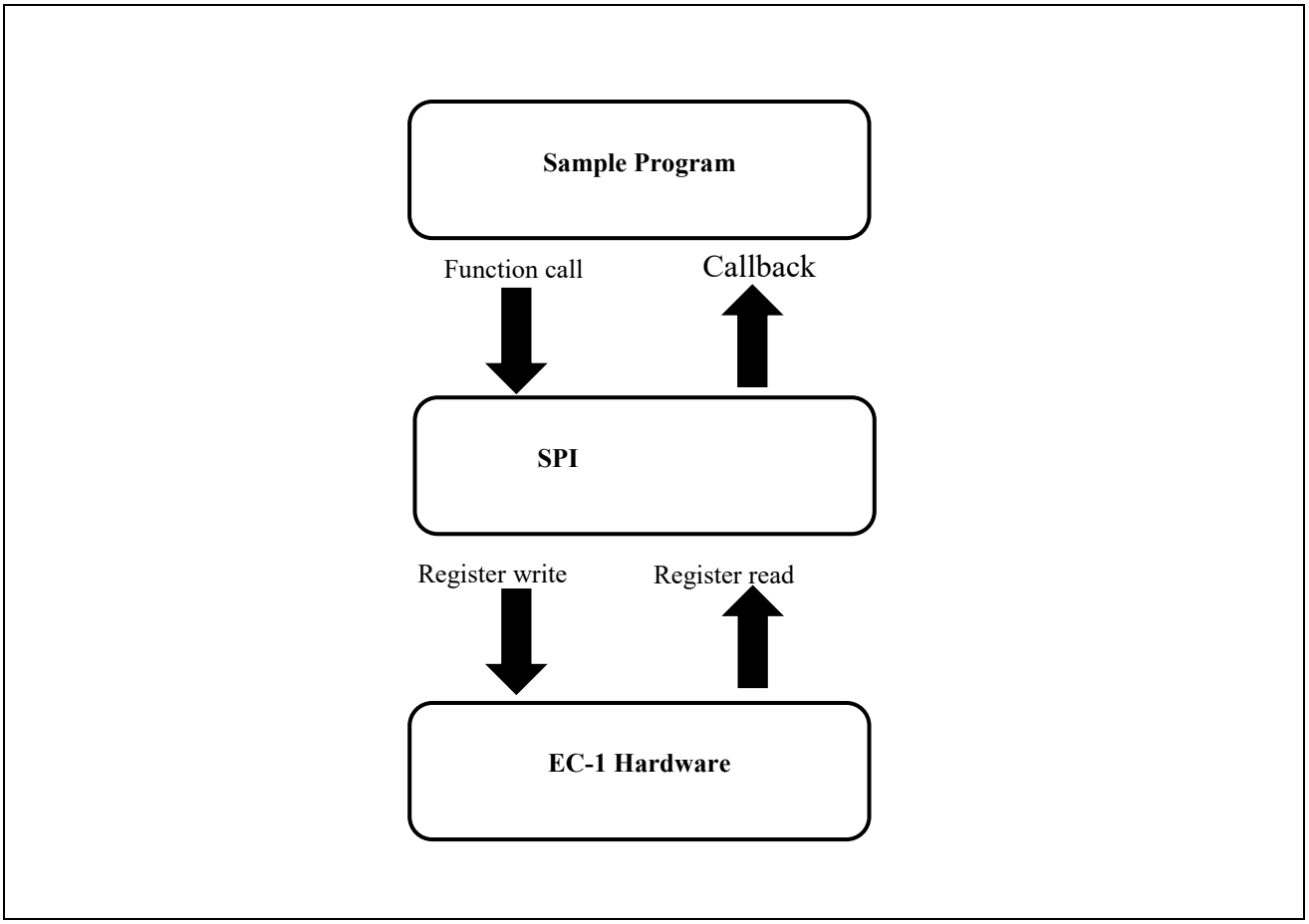


Figure 5-3 System Block Diagram

5.5.2 Preparations

To run this sample program, prepare two boards (master and slave) and connect them as shown in Figure 5-4.

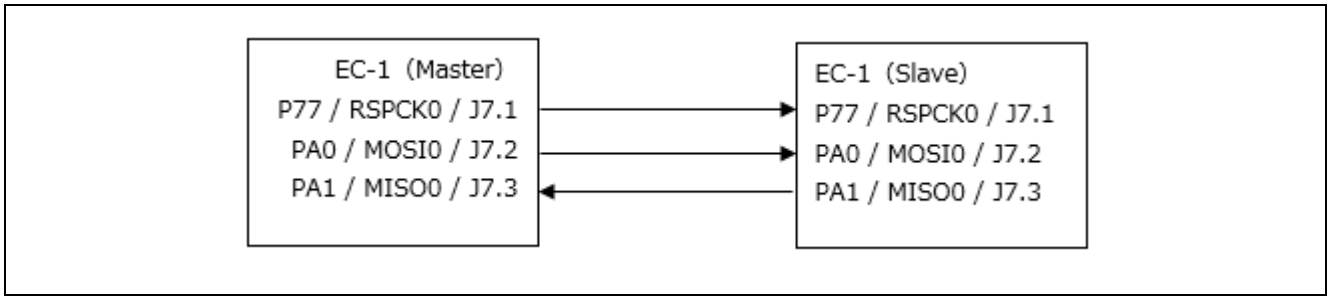


Figure 5-4 Connection

5.5.3 Terminal Software (Tera Term)

This sample program performs RS-232C COM port communications with the host computer by using asynchronous communications of the SCIFA. The settings of the terminal software in the host computer are as follows:

- Transfer rate: 115,200 bps
- Character length: 8 bits
- Stop bit length: 1 bit
- Parity function: None
- Hardware flow control: None

5.5.4 Functions of the Sample Program

- Switches between the master and slave according to the DIP switch (SW3-8) setting at the time of startup.
- Performs slave reception and master transmission by operating the DIP switch (SW3-1).
- Displays part of the data for transmission and received data at the UART.

5.5.5 Sample Program Execution Example

(1) Set SW2-8 on the Evaluation board to ON and set the slave to OFF.

(2) Connect the USB virtual COM port (CN4) on the Evaluation board to the host PC, run the terminal software, and then execute the sample program.

(3) Turn SW3-1 on in the slave to place the slave in the received data waiting state.

(4) When SW3-1 is turned on in the master, the master starts transmitting data.

Data is transmitted and received between the master and the slave and the results are displayed as a log.

(5) To continue data transfer, perform steps (3) and (4) after turning off SW3-1 in the master and slave.

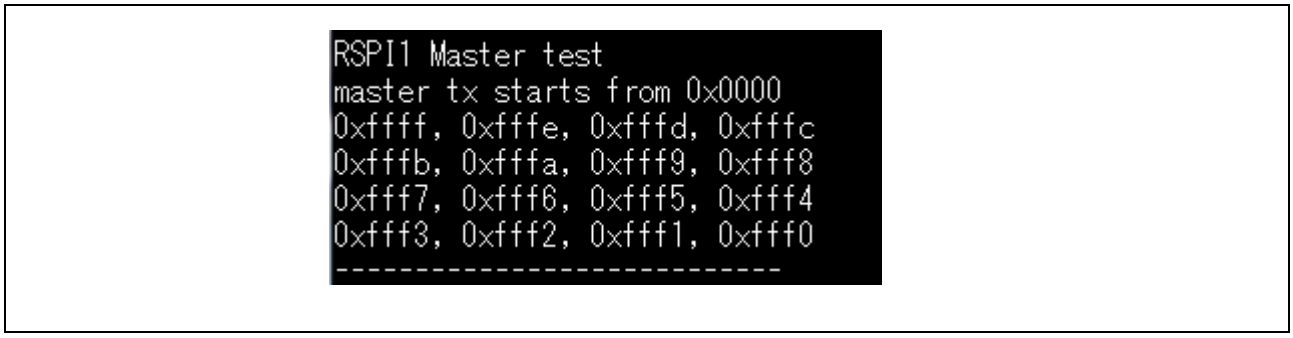


Figure 5-5 Example of Master Test Execution Result

“master tx starts from 0x0000” indicates that the master sent out 16-word increment data starting from 0x0000.

The next 16 words (0xffff to 0xfff0) show the results of transmitted data reception from the slave.

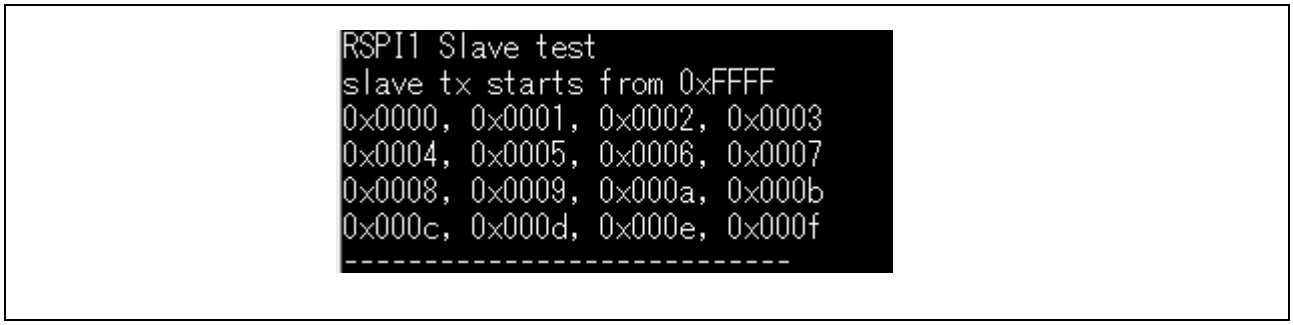


Figure 5-6 Example of Slave Test Execution Result

“slave tx starts from 0xFFFF” indicates that the slave sent out 16-word increment data starting from 0xFFFF.

The next 16 words (0x0000 to 0x000f) show the results of transmitted data reception from the master.

6. SFLASH_WRITER Sample Software

6.1 Overview

This section describes the sample program that uses the serial flash ROM driver that controls the serial flash ROM installed on the evaluation board.

The features of the SFLASH_WRITER sample program are as follows.

Read, write and erase from or to the serial flash ROM can be tested by connecting to a personal computer and by using terminal software.

Target Devices

EC-1

When applying the sample program covered in this application note to another microcontroller, modify the program according to the specifications for the target microcontroller and conduct an extensive evaluation of the modified program.

6.2 Constants

Table 6-1 shows the constants used in this sample code.

Table 6-1 Constants Used in the Sample Program

Constant Name	Setting Value	Description
CMDBUF_MAX_SIZE	256	Command buffer size
RECIEVEBUF_MAX_SIZE	256*1024	Maximum reception buffer size
EXT_RAM_ADDR	&recv_buf[0][0]	Data save address
IMG_SRC_AADR	(uint32_t)EXT_RAM_ADDR	Data save address
IMG_SRC_SIZE	RECIEVEBUF_MAX_SIZE	Maximum reception buffer size
ER_OK	(ER_RET)0	Normal end (no error)
ER_NG	(ER_RET)-1	Abnormal end (error)
ER_SYS	(ER_RET)(2 * ER_NG)	Undefined error
ER_PARAM	(ER_RET)(3 * ER_NG)	Invalid parameter
ER_NOTYET	(ER_RET)(4 * ER_NG)	Incomplete processing
ER_NOMEM	(ER_RET)(5 * ER_NG)	Out of memory
ER_BUSY	(ER_RET)(6 * ER_NG)	Busy
ER_INVAL	(ER_RET)(7 * ER_NG)	Invalid state
ER_TIMEOUT	(ER_RET)(8 * ER_NG)	Timeout occurs

6.3 Structures, Unions and Enumerated Types

The following shows the structures, unions, and enumerated types used in this sample code.

Table 6-2 writer_command_info_t Structure

Member Name	Description
uint8_t* name;	Command name
ER_RET (*func)(int32_t, uint8_t**);	Command function
uint8_t* help;	Command help

6.4 Functions

A list of functions is shown in Table 6-3 List of Functions.

Table 6-3 List of Functions

Function Name	Description	Scope	Definition File
main	Main processing	global	main.c
port_init	Initializes the port settings	local	main.c
flash_writer	flash_writer main processing	global	flash_writer.c
exec_getline	Reads the input character string	local	flash_writer.c
exec_command	Executes the command	local	flash_writer.c
exec_cmd_help	Executes the help command	local	flash_writer.c
exec_cmd_sfw	Executes the sfw command	local	flash_writer.c
exec_cmd_sfr	Executes the sfr command	local	flash_writer.c
exec_cmd_sfe	Executes the sfe command	local	flash_writer.c
exec_cmd_flash_info	Executes the flash_info command	local	flash_writer.c
exec_flash_write	Writes data to the flash	local	flash_writer.c
exec_flash_read	Reads data from the flash	local	flash_writer.c
exec_flash_erase	Erases data from the flash	local	flash_writer.c
btld_flash_write	Writes data to the flash	local	flash_writer.c
btld_flash_erase	Erases data from the flash	local	flash_writer.c
hex2dec	Converts between decimal and hexadecimal	local	flash_writer.c
*dmac_memcpy	dmac_memcpy function	local	flash_writer.c

6.5 Details of the Functions

6.5.1 main

(1) **Synopsis**

Main processing of the sample program

(2) **C language format**

void main (void);

(3) **Parameters**

None

(4) **Function**

The flowchart of the main processing is shown in section 6.6.1, Main processing.

- Initialize the ECM.
- Initialize the port settings.
- Set the SCIFA0.
- Call the flash_writer() program.

(5) **Returned values**

None

6.5.2 port_init

(1) **Synopsis**

Initializes the port settings.

(2) **C language format**

static void port_init (void);

(3) **Parameters**

None

(4) **Function**

This function initializes the port settings.

(5) **Returned values**

None

6.5.3 flash_writer

(1) Synopsis

flash_writer processing

(2) C language format

void flash_writer(void);

(3) Parameters

None

(4) Function

This flash_writer processing is called by the main() processing.

This function accepts an input at the terminal software and performs processing according to the input command.

Figure 6-2 is the flowchart of this function.

(5) Returned values

None

6.5.4 exec_getline

(1) Synopsis

Reads the input character string.

(2) C language format

static ER_RET exec_getline(uint8_t* str, uint32_t echo);

(1) Parameters

I/O	Parameter	Description
O	uint8_t* str	Buffer pointer
I	uint32_t echo	0: Echoback Other than 0: Echoback

(2) Function

This function waits for an input.

It reads the character string specified by the argument until the delimiter or end of the character string.

(3) Returned values

Returned Value	Meaning
ER_NG	Buffer overflow
ER_NOTYET	End of character string not detected
ER_OK	End of character string detected

6.5.5 exec_command

(1) **Synopsis**

Executes the command.

(2) **C language format**

static ER_RET exec_command(uint8_t* str);

(3) **Parameters**

I/O	Parameter	Description
I	uint8_t* str	Pointer to the command buffer

(4) **Description**

This function executes the specified processing when the start of the space-delimited character string passed as the argument matches the supported command.

(5) **Returned values**

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.6 exec_help

(1) Synopsis

Executes the help command.

(2) C language format

static ER_RET exec_cmd_help(int32_t argc, uint8_t* argv[]);

(3) Parameters

I/O	Parameter	Description
I	int32_t argc	Number of arguments
I	uint8_t* argv[]	Argument character string (command name)

(4) Description

This processing is performed when “help” is input in the terminal software.

A list of commands will be displayed.

For details, see Figure 6-6 Example of Help Command Execution Results.

(5) Returned values

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.7 exec_cmd_sfw

(1) Synopsis

Executes the sfw command.

(2) C language format

static ER_RET exec_cmd_sfw(int32_t argc, uint8_t* argv[]);

(3) Parameters

I/O	Parameter	Description
I	int32_t argc	Number of arguments
I	uint8_t* argv[]	Argument character string (command name, address and data)

(4) Description

This processing is performed when “sfw” is input in the terminal software.

This function calls the exec_flash_write() processing and writes data to the flash.

For details, see Figure 6-7 Example of Flash Write Command Execution Results.

(5) Returned values

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.8 exec_cmd_sfr

(1) **Synopsis**

Executes the sfr command.

(2) **C language format**

static ER_RET exec_cmd_sfr(int32_t argc, uint8_t* argv[]);

(3) **Parameters**

I/O	Parameter	Description
I	int32_t argc	Number of arguments
I	uint8_t* argv[]	Argument character string (command name, address and data length)

(4) **Description**

This processing is performed when “sfr” is input in the terminal software.

This function calls the exec_flash_read() processing and reads data from the flash.

For details, see Figure 6-8 Example of Flash Read Command Execution Results.

(5) **Returned values**

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.9 exec_cmd_sfe

(1) **Synopsis**

Executes the sfe command.

(2) **C language format**

static ER_RET exec_cmd_sfe(int32_t argc, uint8_t* argv[]);

(3) **Parameters**

I/O	Parameter	Description
I	int32_t argc	Number of arguments
I	uint8_t* argv[]	Argument character string (command name and address)

(4) **Description**

This processing is performed when “sfe” is input in the terminal software.

This function calls the exec_flash_erace() processing and erases the specified sector of the flash.

For details, see Figure 6-9 Example of Flash Erase Command Execution Results.

(5) **Returned values**

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.10 exec_cmd_flash_info

(1) **Synopsis**

Executes the flash_info command.

(2) **C language format**

static ER_RET exec_cmd_flash_info(int32_t argc, uint8_t* argv[]);

(3) **Parameters**

I/O	Parameter	Description
I	int32_t argc	Number of arguments
I	uint8_t* argv[]	Argument character string

(4) **Description**

This processing is performed when “flash_info” is input in the terminal software.

It displays the device information.

For details, see Figure 6-10 Example of Flash Device Information Display Command Execution Results.

(5) **Returned values**

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.11 exec_flash_write

(1) **Synopsis**

Writes data to the flash.

(2) **C language format**

static ER_RET exec_flash_write(int32_t argc, uint8_t* argv[]);

(3) **Parameters**

I/O	Parameter	Description
I	int32_t argc	Number of arguments
I	uint8_t* argv[]	Argument character string (command name, address and data)

(4) **Description**

This function verifies the parameters and sectors and writes to the flash.

(5) **Returned values**

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.12 exec_flash_read

(1) **Synopsis**

Reads data from the flash.

(2) **C language format**

static ER_RET exec_flash_read(int32_t argc, uint8_t* argv[]);

(3) **Parameters**

I/O	Parameter	Description
I	int32_t argc	Number of arguments
I	uint8_t* argv[]	Argument character string (command name, address and data length)

(4) **Description**

This function checks the parameters and reads data from the flash.

(5) **Returned values**

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.13 exec_flash_erase

(1) **Synopsis**

Erases data from the flash.

(2) **C language format**

static ER_RET exec_flash_erase(int32_t argc, uint8_t* argv[]);

(3) **Parameters**

I/O	Parameter	Description
I	int32_t argc	Number of arguments
I	uint8_t* argv[]	Argument character string (command name, address and data length)

(4) **Description**

This function verifies the parameters and sectors, and erases the specified flash area.

(5) **Returned values**

Returned Value	Meaning
ER_OK	No error
ER_NG	Error

6.5.14 btld_flash_write

(1) **Synopsis**

Writes data to the flash.

(2) **C language format**

static ER_RET btld_flash_write(uint16_t* buf, uint32_t addr, uint32_t size);

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t* buf	Buffer address
I	uint32_t addr	Program base address
I	uint32_t size	Program size

(4) **Description**

This function writes, in big endian, 1- to 16-byte data to the position specified by the arguments in the serial flash ROM.

(5) **Returned values**

Returned Value	Meaning
ER_OK	Success
ER_NG	Parameter error

6.5.15 btld_flash_erace

(1) **Synopsis**

Erases data from the flash.

(2) **C language format**

static ER_RET btld_flash_erase(uint32_t addr, uint32_t size);

(3) **Parameters**

I/O	Parameter	Description
I	uint32_t addr	Program base address
I	uint32_t size	Program size

(4) **Description**

This function erases a sector which includes the specified address.

(5) **Returned values**

Returned Value	Meaning
ER_OK	Success
ER_NG	Parameter error

6.5.16 hex2dec

(1) **Synopsis**

Converts between decimal and hexadecimal.

(2) **C language format**

static ER_RET hex2dec(uint8_t* str);

(3) **Parameters**

I/O	Parameter	Description
I	uint8_t* str	Hexadecimal character string

(4) **Description**

This function converts the hexadecimal character string specified by the argument to a decimal value.

(5) **Returned values**

Returned Value	Meaning
ER_OK	Success
ER_NG	Parameter error

6.5.17 dmac_memcpy

(1) **Synopsis**

dmac_memcpy function

(2) **C language format**

static void *dmac_memcpy(void *dst, const void *src, uint32_t n);

(3) **Parameters**

I/O	Parameter	Description
I	void *dst	Destination address
I	const void *src	Copy source address
I	uint32_t n	Number of copied bytes

(4) **Description**

This function accesses the memory directly and performs memory copy.

(5) **Returned values**

Returned Value	Meaning
ER_OK	Success
ER_NG	Parameter error

6.6 Flowcharts

6.6.1 Main processing

The figure below is a flowchart of the main processing of the sample code.

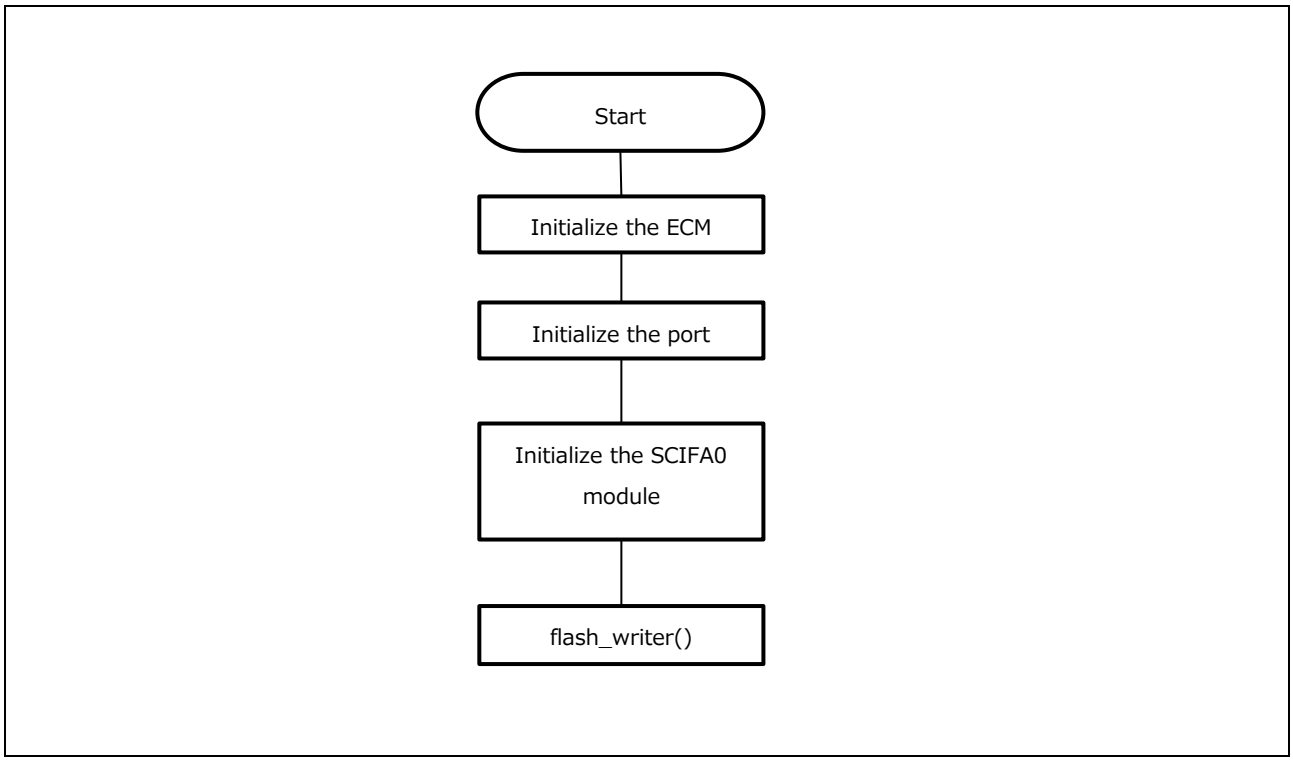


Figure 6-1 Main Processing of the Sample Code

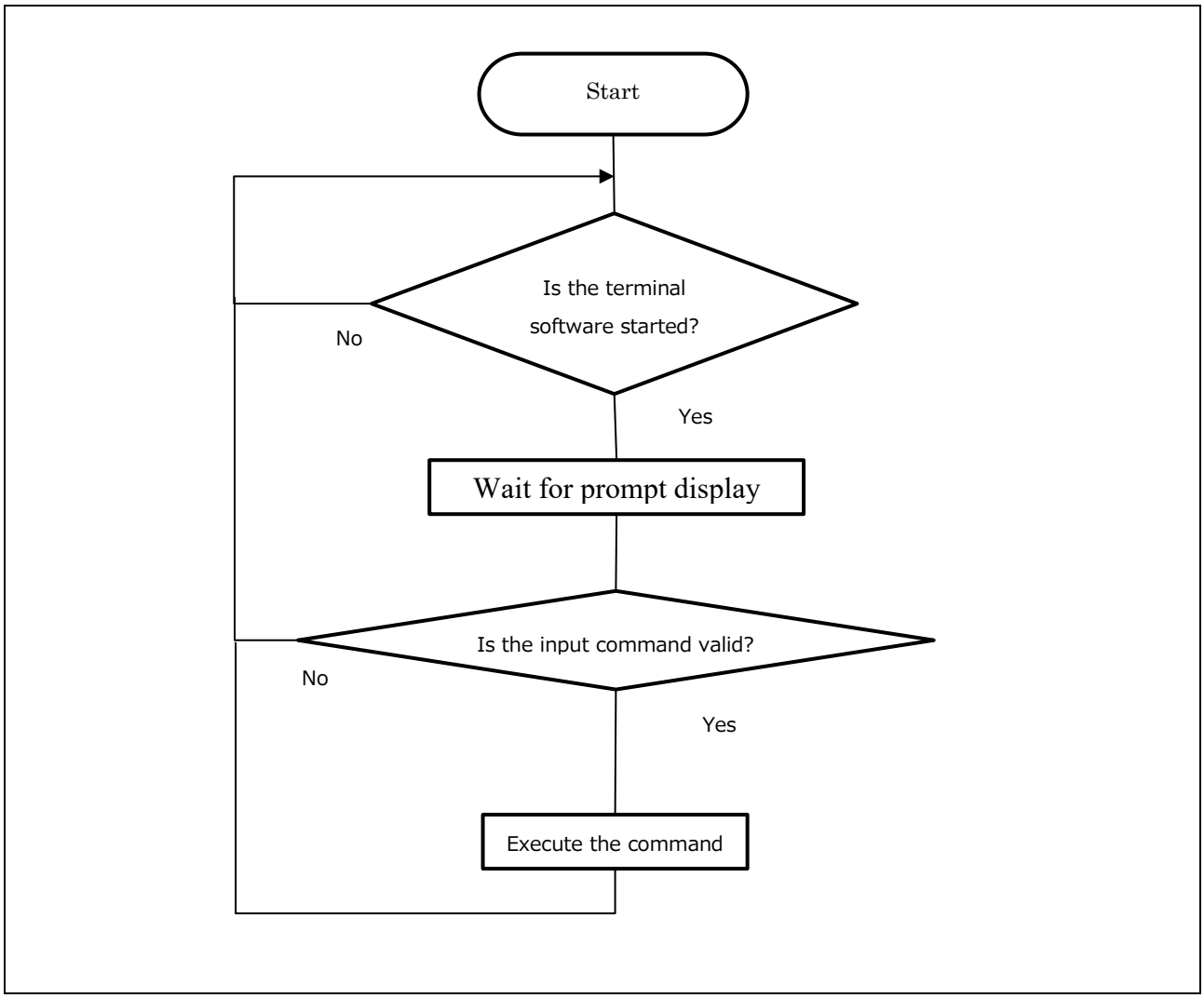


Figure 6-2 flash_writer() Processing

6.7 Tutorials

6.7.1 Operational Overview

Table 6-4 Operational Overview summarizes the functions of the flash_writer sample program. Figure 6-3 shows a system block diagram.

Table 6-4 Operational Overview

Function	Description
Communication channel	Channel 0 (SCIFA0) is used.
Serial communication	Asynchronous
Clock	SERICLK = 150 MHz
Transmission/reception	Serial data transmission/reception
Transfer rate	115,200 bps
Character length	8 bits
Stop Bit Length	1 bits
Parity function	None
Hardware flow control	None
Operation	Operation is selected by the command.
Operation result display	The result is output to the console.

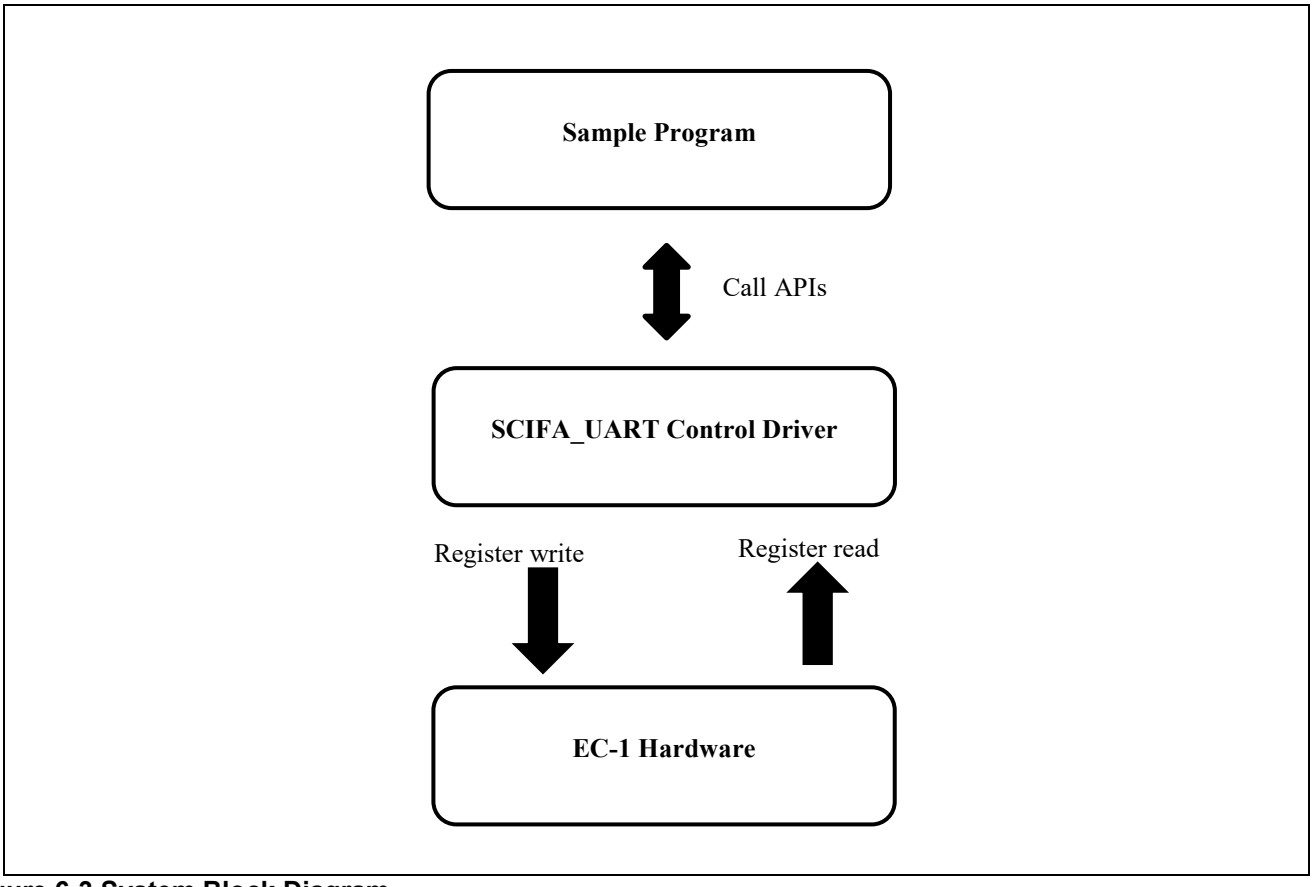


Figure 6-3 System Block Diagram

6.7.2 Preparations

This sample program performs RS-232C COM port communications with the host computer by using asynchronous communications of the SCIFA. For the communication settings, see section 6.7.3, Terminal Software (Tera Term).

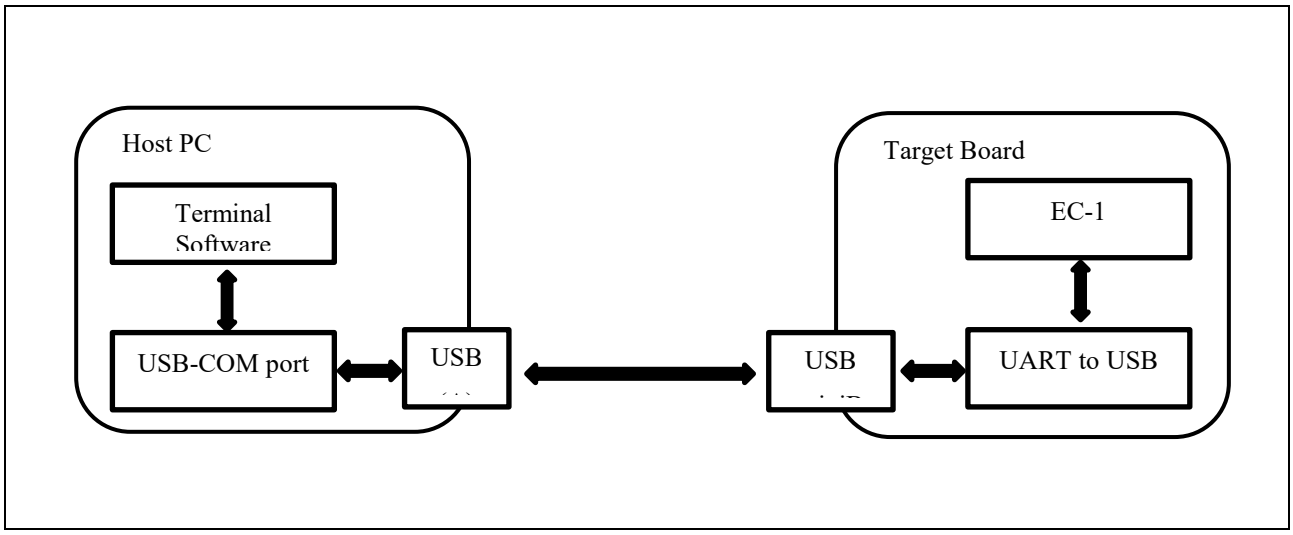


Figure 6-4 Connection

6.7.3 Terminal Software (Tera Term)

The settings of the terminal software in the host computer are as follows:

- Transfer rate: 115,200 bps
- Character length: 8 bits
- Stop bit length: 1 bit
- Parity function: None
- Flow control: Not provided

6.7.4 Functions of the Sample Program

Start the terminal software (Tera Term), and then start this sample program.

A command from the host computer is accepted when command prompt EC-1> as shown in Figure 6-5 is displayed.

Figure 6-5 Display When the Sample Program Is Started

Table 6-5 lists the commands.

Table 6-5 List of Commands

Format	Description	Control
help	Help display	Displays the list of control commands.
sfw [address][data]	Serial flash write	Writes N-bytes of [data] starting from [address] to the serial flash*1, *2.
sfr [address][length]	Serial flash read	Reads [length] bytes starting from [address] in serial flash*1.
sfe [address]	Serial flash erase	Erases one sector or all sectors starting from [address] in serial flash*1.
flash_info	Flash device information display	Displays the device information about the serial flash.

Note 1. Specify in hexadecimal without “0x”.

Note 2. A variable length of data from 1 to 16 bytes can be specified for [data].

6.7.5 Sample Program Execution Example

- Help command (help)

Displays a list of control commands.

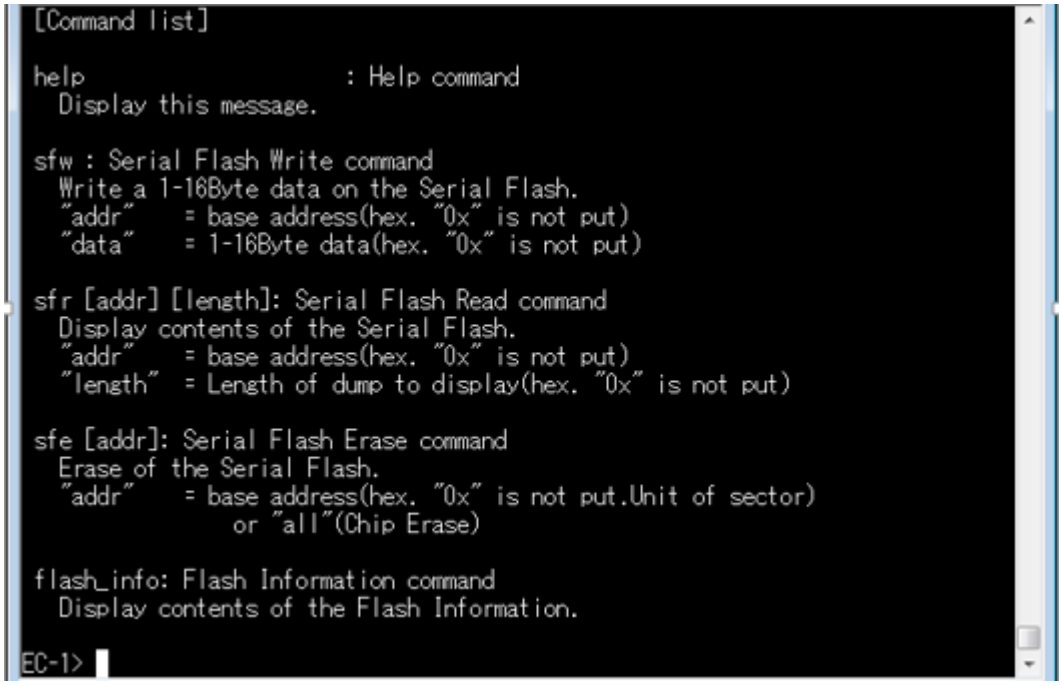


Figure 6-6 Example of Help Command Execution Results

- Flash write command (sfw)

Writes, in big endian, 1- to 16-byte data to an arbitrary position in the flash.

Figure 6-7 Example of Flash Write Command Execution Results

- Flash read command (sfr)

Displays the content of the specified number of bytes from the specified position in the flash.

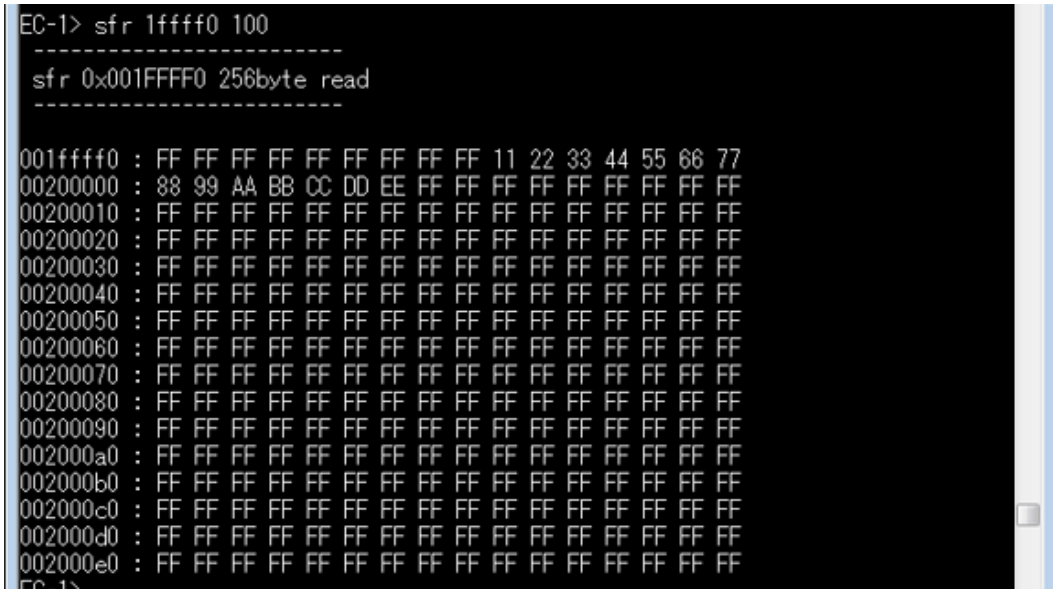


Figure 6-8 Example of Flash Read Command Execution Results

- Flash erase command (sfe)

Erases the sector which includes the specified address.

If “all” is specified for the address, all sectors are erased.

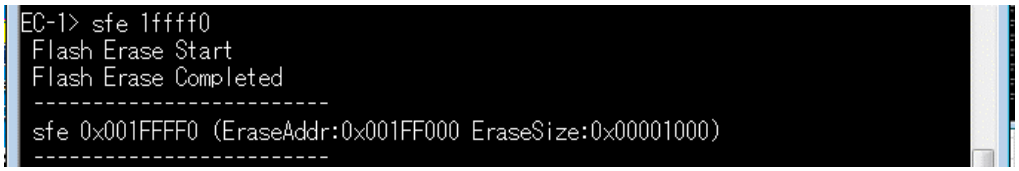


Figure 6-9 Example of Flash Erase Command Execution Results

- Flash device information display (flash_info)

Displays the device information about the serial flash.

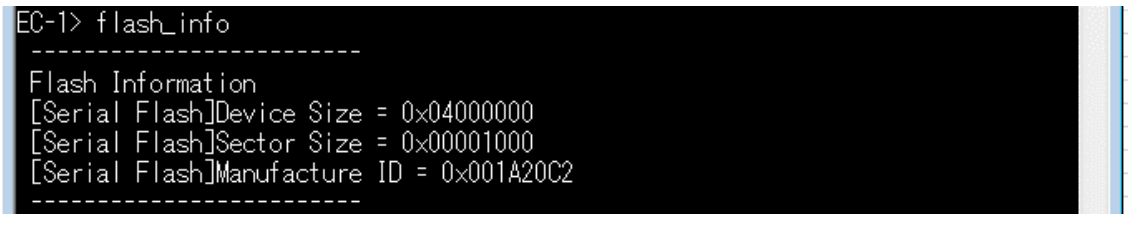


Figure 6-10 Example of Flash Device Information Display Command Execution Results

7. USB Function Sample Software

7.1 Overview

This section describes the sample program that uses the USB function driver that controls the USB function installed on the evaluation board.

The features of the USB function sample program are as follows.

The USB function driver is a combination of the following modules (PCD and PCDC).

(1) Peripheral control driver (PCD)

The PCD controls the hardware of the USB2.0HS function module. The PCD operates by combining with the sample device class driver that Renesas provides or the peripheral device class driver (PDCD) that the user creates.

The functions this module supports are as follows:

- Device attach/detach, suspend/resume, USB bus reset
- Control transfer over pipe 0
- Bulk transfer, interrupt transfer or isochronous transfer (CPU/DMA access selectable) over pipes 1 to 9
- Enumeration with the USB1.1 / 2.0 / 3.0 host

Restrictions

The following restrictions apply to this module:

- The pipe usage is restricted by the pipe information setting function.

  → The receiving pipes use the transaction counter for the SHTNAK function.
- The structure consists of members of different types.

  (The compiler might cause address misalignment of structure members.)
  - Suspension during data transfer is not supported. Execute suspension after making sure that data transfer has finished.

For details, see section 7.8, Peripheral Control Driver (PCD).

For details, see section 7.9, Peripheral Device Class Driver (PDCD).

(2) USB peripheral communications device class driver (PCDC)

Installed as an abstract control model complying with the USB communication device class specifications, the peripheral device class driver (PDCD) can communicate with the USB host when combined with the PCD.

The functions this module supports are as follows:

- Data transfer with the USB host
- Response to CDC class requests
- Communication device class notification transmission service

For details, see section 7.10, Peripheral Communication Device Driver (PCDC).

Glossary

APL: Application program

CDC: Communications Devices Class

H/W: Renesas USB device

PCD: Peripheral control driver

PDCD: Peripheral device class driver (device driver and USB class driver)

PCDC: Peripheral Communications Devices Class

USB: Universal Serial Bus

Target Devices

EC-1

When applying the sample program covered in this application note to another microcontroller, modify the program according to the specifications for the target microcontroller and conduct an extensive evaluation of the modified program.

7.2 Constants

Table 7-1 shows the constants used in this sample code.

Table 7-1 Constants Used in the Sample Program

Constant Name	Setting Value	Description
CDC_DATA_LEN	512	USB transfer data length
EVENT_MAX	5	Maximum number of events
TASK_LOOPS_BETWEEN_INSTRUCTIONS	0x500000	Address at which the initial connection message is stored
ALIGN_SIZE	0x20	Alignment size

7.3 Structures, Unions and Enumerated Types

The following shows the structures, unions, and enumerated types used in this sample code.

Table 7-2 DEV_INFO_t Structure

Member Name	Description
uint16_t state;	State for application
uint16_t event_cnt;	Event count
uint16_t event[EVENT_MAX];	Event no.

Table 7-3 STATE_t Enumerated Type

Member Name	Description
STATE_ATTACH;	Attach processing
STATE_DATA_TRANSFER	Data transfer processing
STATE_DETACH	Detach processing

Table 7-4 EVENT_t Enumerated Type

Member Name	Description
EVENT_NONE	No event
EVENT_DETACH	USB device disconnect
EVENT_CONFIGERD	USB device connection completed
EVENT_USB_READ_START	Request to receive USB data
EVENT_USB_READ_COMPLETE	USB data reception completion
EVENT_USB_WRITE_START	Request to send USB data
EVENT_USB_WRITE_COMPLETE	USB data transmission completion
EVENT_COM_NOTIFY_START	Request to send class notification "SerialState"
EVENT_COM_NOTIFY_COMPLETE	Class notification “SerialState” transmission completion

Table 7-5 USB_PCDC_APL_STATE Enumerated Type

Member Name	Description
APP_STATE_IDLE	Idle state
APP_STATE_ECHO_MODE	Echo mode

7.4 Functions

A list of functions is shown in Table 7-6 List of Functions.

Table 7-6 List of Functions

Function Name	Description	Scope	Definition File
main	Main processing	global	main.c
port_init	Initializes the port settings	global	main.c
icu_init	Sets interrupts	global	main.c
usbf_main	USB function main processing	global	r_usb_pcdc_apl.c
cdc_connect_wait	USB connection wait processing	global	r_usb_pcdc_apl.c
cdc_detach_device	Detach processing	global	r_usb_pcdc_apl.c
cdc_data_transfer	Data transfer processing	global	r_usb_pcdc_apl.c
cdc_read_complete	USB reception completion callback function	global	r_usb_pcdc_apl.c
cdc_write_complete	USB transmission completion callback function	global	r_usb_pcdc_apl.c
cdc_demo_complete	Demonstration-procedure transmission completion callback function	global	r_usb_pcdc_apl.c
cdc_configured	Device configured callback function	global	r_usb_pcdc_apl.c
cdc_detach	Detach processing callback function	global	r_usb_pcdc_apl.c
cdc_default	Default processing callback function	global	r_usb_pcdc_apl.c
cdc_suspend	Suspend processing callback function	global	r_usb_pcdc_apl.c
cdc_resume	Resume processing callback function	global	r_usb_pcdc_apl.c
cdc_interface	Interface processing callback function	global	r_usb_pcdc_apl.c
cdc_registration	Registers the device driver	global	r_usb_pcdc_apl.c
apl_init	Initializes the APL	global	r_usb_pcdc_apl.c
cdc_event_set	Issues an event	global	r_usb_pcdc_apl.c
cdc_event_get	Gets an event	global	r_usb_pcdc_apl.c

7.5 Details of the Functions

7.5.1 main

(1) **Synopsis**

Main processing of the sample program

(2) **C language format**

void main (void);

(3) **Parameters**

None

(4) **Description**

This function is the main processing of sample program:

- Initialize the ECM.
- Set interrupts.
- Initialize the port settings.
- Set the SCIFA0.
- Call the usbf_main() program.

(5) **Returned values**

None

7.5.2 port_init

(1) **Synopsis**

Initializes the port settings.

(2) **C language format**

void port_init (void);

(3) **Parameters**

None

(4) **Description**

This function initializes the port settings.

(5) **Returned values**

None

7.5.3 icu_init

(1) **Synopsis**

Sets interrupts.

(2) **C language format**

void icu_init (void);

(3) **Parameters**

None

(4) **Description**

This function enables interrupt processing.

(5) **Returned values**

None

7.5.4 usbf_main

(1) Synopsis

USB function main processing

(2) C language format

void usbf_main (void);

(3) Parameters

None

(4) Description

This function is the main processing of sample program.

Figure 7-1 is the flowchart of the main processing.

It manages the USB function according to the state and the event associated with the state.

It checks for an event associated with the state and performs processing according to the event.

After processing the event, the application changes its state as required.

- STATE_ATTACH: Attach processing
- STATE_DATA_TRANSFER: Data transfer processing
- STATE_DETACH: Detach processing

(5) Returned values

None

7.5.5 cdc_connect_wait

(1) Synopsis

USB connection wait processing

(2) C language format

uint16_t cdc_connect_wait(void);

(3) Parameters

None

(4) Description

This function is called in the STATE_ATTACH state and when the USB host is not connected. When enumeration with the USB host is completed, callback function cdc_configured() issues event EVENT_CONFIGURD.

EVENT_CONFIGURD changes the state to STATE_DATA_TRANSFER and causes EVENT_USB_READ_START to be issued.

For details, see section 7.6.2, Management of States and Events.

(4) Returned values

Returned Value	Meaning
USB_TRUE	Success
USB_FALSE	Error

7.5.6 cdc_detach_device

(1) Synopsis

Detach processing

(2) C language format

void cdc_detach_device(void);

(3) Parameters

None

(4) Description

This function is called in the STATE_DETACH state.
It clears the variables and changes the state to STATE_ATTACH.
For details, see section 7.6.2, Management of States and Events.

(5) Returned values

None

7.5.7 cdc_deta_transfer

(1) Synopsis

Data transfer processing

(2) C language format

uint16_t cdc_data_transfer(void);

(3) Parameters

None

(4) Description

This function is called in the STATE_DATA_TRANSFER state.

It sends an initial connection message to the USB host at specified intervals until it receives an initial message from the USB host.

After receiving the initial data, it performs loopback processing which receives data from the USB host and sends the received data to the USB host as is.

For details, see section 7.6.2, Management of States and Events.

(6) Returned values

Returned Value	Meaning
USB_TRUE	Success
USB_FALSE	Error

7.5.8 cdc_read_complete

(1) **Synopsis**

USB reception completion callback function

(2) **C language format**

void cdc_read_complete(USB_UTR_t *mess);

(3) **Parameters**

I/O	Parameter	Description
I	USB_UTR_t *mess	Message

(4) **Description**

This callback function is called when data reception processing is completed in the STATE_DATA_TRANSFER state.

It issues EVENT_USB_READ_COMPLETE.

(5) **Returned values**

None

7.5.9 cdc_write_complete

(1) **Synopsis**

USB transmission completion callback function

(2) **C language format**

void cdc_write_complete(USB_UTR_t *mess);

(3) **Parameters**

I/O	Parameter	Description
I	USB_UTR_t *mess	Message

(4) **Description**

This callback function is called when data transmission processing is completed in the STATE_DATA_TRANSFER state.

It issues EVENT_USB_WRITE_COMPLETE.

(5) **Returned values**

None

7.5.10 cdc_demo_complete

(1) **Synopsis**

Demonstration-procedure transmission completion callback function

(2) **C language format**

void cdc_demo_complete(USB_UTR_t *mess);

(3) **Parameters**

I/O	Parameter	Description
I	USB_UTR_t *mess	Message

(4) **Description**

This callback function is called each time an initial connection message is sent to the USB host at specified intervals in the STATE_DATA_TRANSFER state.

(5) **Returned values**

None

7.5.11 cdc_configured

(1) **Synopsis**

Device configured callback function

(2) **C language format**

void cdc_configured(uint16_t data1);

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t data1	This argument is not used.

(4) **Description**

This callback function is called by the PCD when the USB state changes.

It performs processing for the configured state.

This callback function is called when enumeration is completed in the STATE_ATTACH state.

It issues EVENT_CONFIGURD.

(5) **Returned values**

None

7.5.12 cdc_detach

(1) **Synopsis**

Detach processing callback function

(2) **C language format**

void cdc_detach(uint16_t data1);

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t data1	This argument is not used.

(4) **Description**

This callback function is called by the PCD when the USB state changes.

It performs processing for the detach state.

It changes the state to STATE_DETACH.

(5) **Returned values**

None

7.5.13 cdc_default

(1) **Synopsis**

Default processing callback function

(2) **C language format**

void cdc_default(uint16_t mode);

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t mode	Operating device speed USB_HSCONNECT: High-speed USB_FSCONNECT: Full-speed USB_NOCONNECT: No connect

(4) **Description**

This callback function is called by the PCD when the USB state changes.

It performs processing for the default state.

It makes settings for processing USB bus resets.

(5) **Returned values**

None

7.5.14 cdc_suspend

(1) **Synopsis**

Suspend processing callback function

(2) **C language format**

void cdc_suspend(uint16_t data1);

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t mode	This argument is not used.

(4) **Description**

This callback function is called by the PCD when the USB state changes.

It performs processing for the suspend state.

(5) **Returned values**

None

7.5.15 cdc_resume

(1) **Synopsis**

Resume processing callback function

(2) **C language format**

void cdc_resume(uint16_t data1);

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t data1	This argument is not used.

(4) **Description**

This callback function is called by the PCD when the USB state changes.

It performs processing for the resume state.

(5) **Returned values**

None

7.5.16 cdc_interface

(1) **Synopsis**

Interface processing callback function

(2) **C language format**

void cdc_interface(uint16_t data1);

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t data1	This argument is not used.

(4) **Description**

This callback function is called by the PCD when the USB state changes.

It performs processing for the interface state.

(5) **Returned values**

None

7.5.17 cdc_registration

(1) **Synopsis**

Registers the device driver.

(2) **C language format**

void cdc_registration(void);

(3) **Parameters**

None

(4) **Description**

This function makes an initial setting for the USB driver.

(5) **Returned values**

None

7.5.18 apl_init

(1) **Synopsis**

Initializes the APL.

(2) **C language format**

void apl_init(void);

(3) **Parameters**

None

(4) **Description**

This function initializes the sample application.

(5) **Returned values**

None

7.5.19 cdc_event_set

(1) **Synopsis**

Issues an event.

(2) **C language format**

void cdc_event_set(uint16_t event);

(3) **Parameters**

I/O	Parameter	Description
I	uint16_t event	Event

(4) **Description**

It issues the event specified in the argument.

(5) **Returned values**

None

7.5.20 cdc_event_get

(1) **Synopsis**

Gets an event.

(2) **C language format**

uint16_t cdc_event_get(void);

(3) **Parameters**

None

(4) **Description**

It gets an event and returns it as the returned value.

(5) **Returned values**

Returned Value	Meaning
	Event

7.6 Flowcharts

7.6.1 Application (APL)

The APL performs the following processing:

(1) The APL manages the USB function according to the state and the event associated with the state.

It first checks the state.

The state is stored in a member of the structure shown in Table 7-2 DEV_INFO_t Structure that the APL manages.

(2) The APL checks for an event associated with the state and performs processing according to the event.

After processing the event, the APL changes its state as required.

The event is stored in a member of the structure shown in Table 7-2 DEV_INFO_t Structure that the APL manages.

The figure below outlines the processing by the APL.

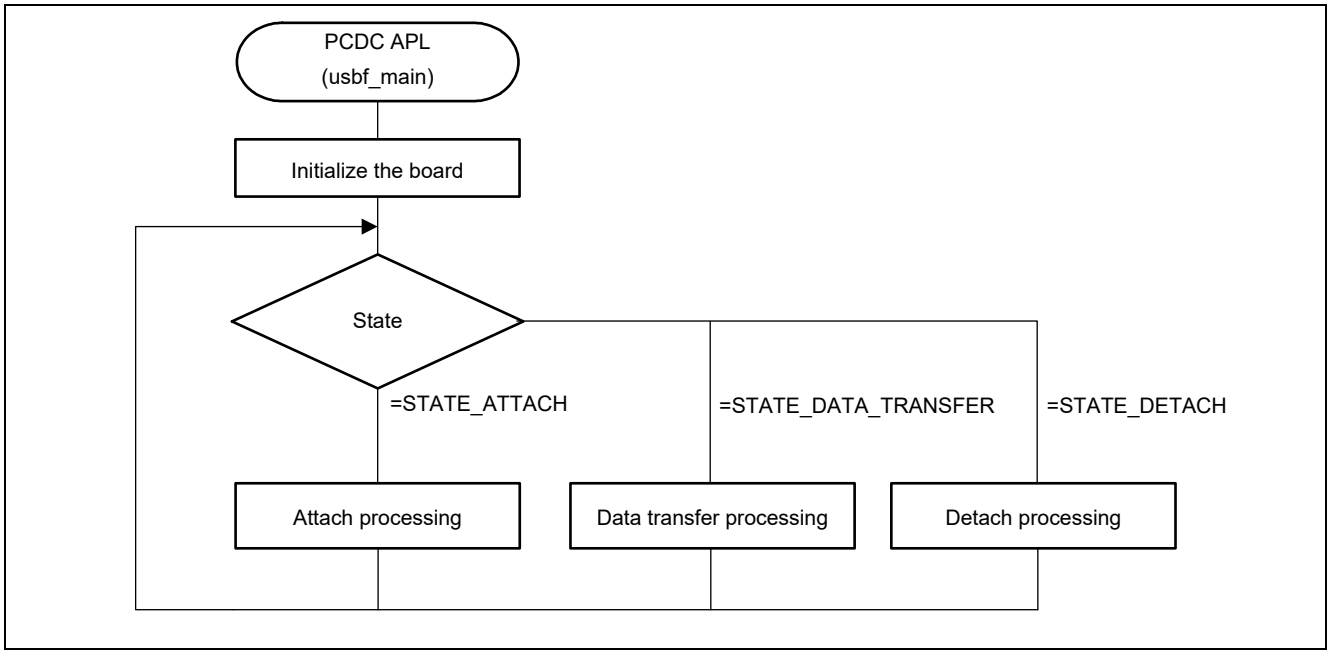


Figure 7-1 Main Processing of the Sample Code

Table 7-7 List of States

State	State Processing Overview	Associated Event
STATE_ATTACH	Attach processing	EVENT_CONFIGURD
STATE_DATA_TRANSFER	Data transfer processing	EVENT_USB_READ_START
		EVENT_USB_READ_COMPLETE
		EVENT_USB_WRITE_START
		EVENT_USB_WRITE_COMPLETE
		EVENT_COM_NOTIFY_START
		EVENT_COM_NOTIFY_COMPLETE
STATE_DETACH	Detach processing	—

Table 7-8 List of Events

Event	Description
EVENT_CONFIGURD	USB device connection completed
EVENT_USB_READ_START	Request to receive USB data
EVENT_USB_READ_COMPLETE	USB data reception completion
EVENT_USB_WRITE_START	Request to send USB data
EVENT_USB_WRITE_COMPLETE	USB data transmission completion
EVENT_COM_NOTIFY_START	Request to send class notification "SerialState"
EVENT_COM_NOTIFY_COMPLETE	Class notification “SerialState” transmission completion
EVENT_NONE	No event

7.6.2 Management of States and Events

States and events are managed by the structure shown below. This structure is provided by the APL.

“EVENT_NONE” is set if no event to be obtained exists in the get event processing.

```
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. */
}
DEV_INFO_t;
```

The following pages outline the processing for each state.

(1) Attach processing (STATE_ATTACH)

In this state, the sample code connected to the USB host notifies the APL of completion of enumeration and changes the state to STATE_DATA_TRANSFER.

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

(2) The STATE_ATTACH state is maintained and cdc_connect_wait() is called until the sample code is connected to the USB host. When enumeration with the USB host is completed after being connected to the USB host, the USB driver calls callback function cdc_configured(), which issues event EVENT_CONFIGURD. This callback function is a function set in member devconfig in the USB_PCDREG_t structure.

(3) EVENT_CONFIGURD changes the state to STATE_DATA_TRANSFER and causes EVENT_USB_READ_START to be issued.

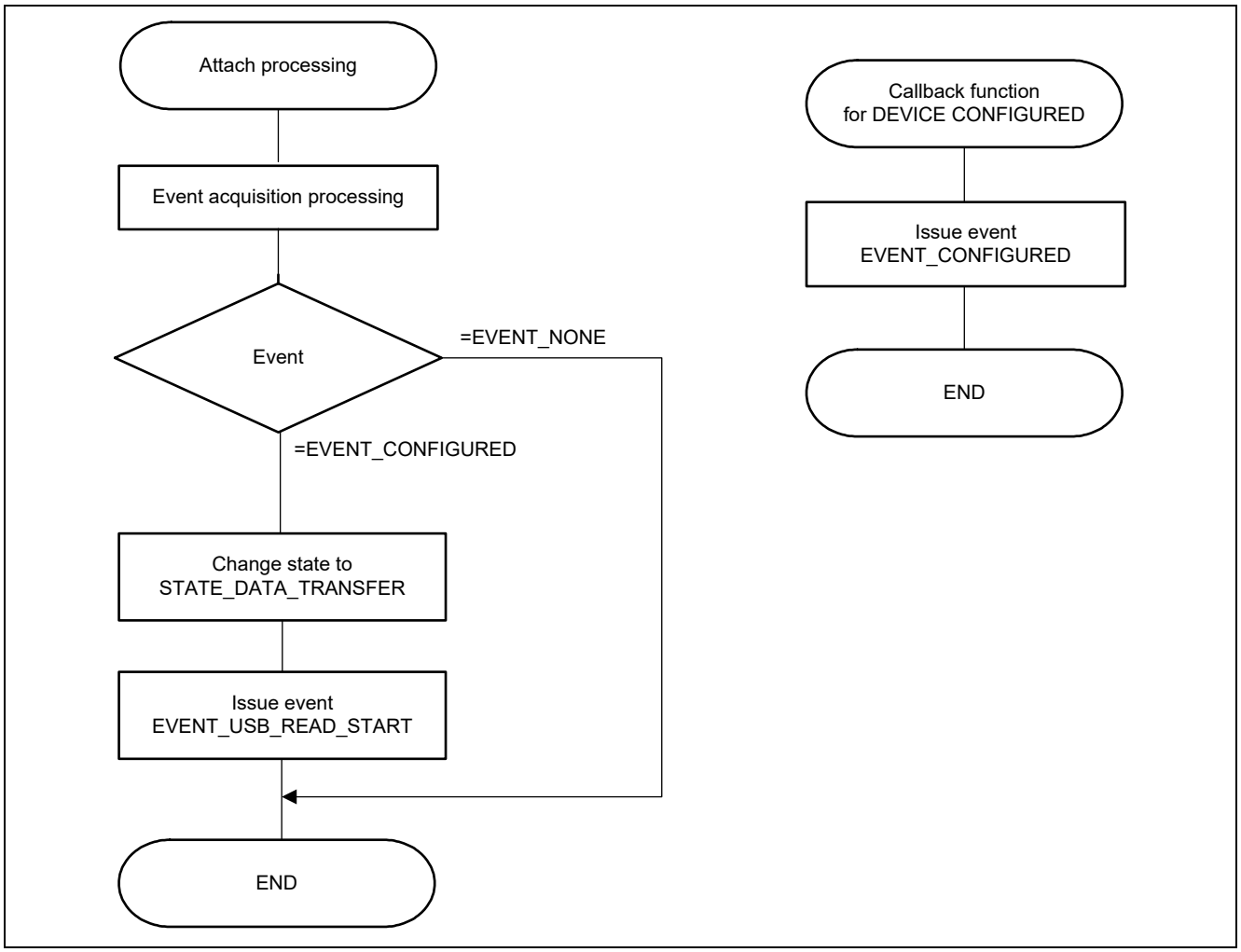


Figure 7-2 Attach Processing of the Sample Code

(2) Data transfer processing (STATE_DATA_TRANSFER)

In this state, the sample code sends an initial connection message to the USB host at specified intervals until it receives an initial message from the USB host. After receiving the initial data, it performs loopback processing which receives data from the USB host and sends the received data to the USB host as is.

Initial connection message

(1) After the USB host is connected, the following message is sent to the USB host at specified intervals:

"PCDC.Virtual serial COM port. Type characters into terminal.

The target will receive the characters over USB CDC, then copy them to a USB transmit buffer to be echoed back over USB."

(2) When data is transferred from the terminal software on the computer, "Echo Mode." is displayed on the terminal software. After this display, loopback processing is performed.

Data reception processing

(1) EVENT_USB_READ_START requests the USB driver to receive data sent from the USB host.

(2) When data reception processing is completed, callback function cdc_read_complete() is called. This callback function issues EVENT_USB_READ_COMPLETE.

(3) EVENT_USB_READ_COMLETE issues event EVENT_USB_WRITE_START.

Data transmission processing

(1) EVENT_USB_WRITE_START requests the USB driver to send the received data above to the USB host.

(2) When data transmission processing is completed, callback function cdc_write_complete() is called. This callback function issues EVENT_USB_WRITE_COMPLETE.

(3) EVENT_USB_WRITE_COMPLETE issues event EVENT_USB_READ_START. This starts data reception processing (1) above again.

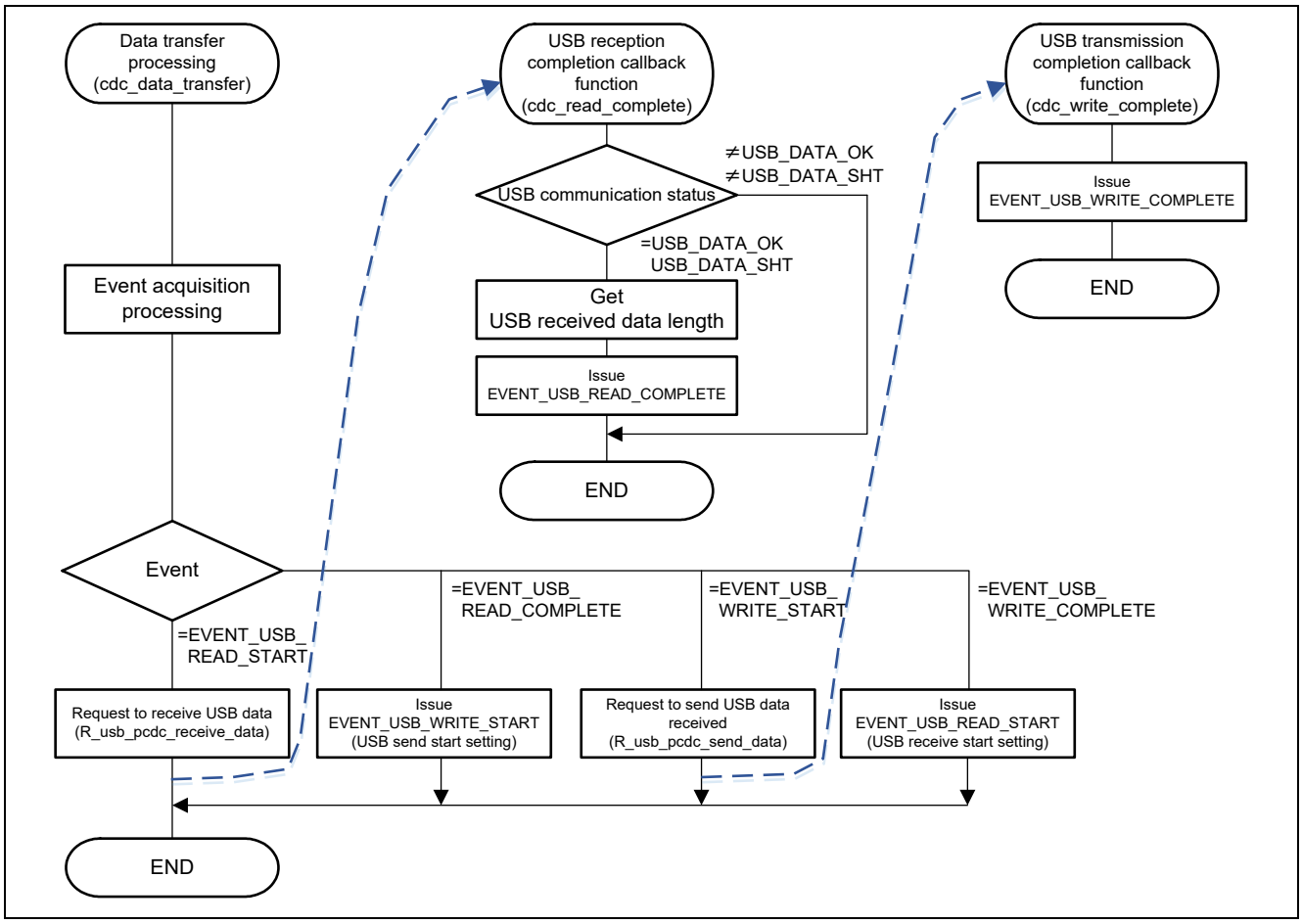


Figure 7-3 Data Transfer Processing of the Sample Code

(3) **Detach processing (STATE_DETACH)**

When disconnected from the USB host, callback function cdc_detach() is called by the USB driver. This callback function changes the state to STATE_DETACH. STATE_DETACH clears the variables and changes the state to STATE_ATTACH.

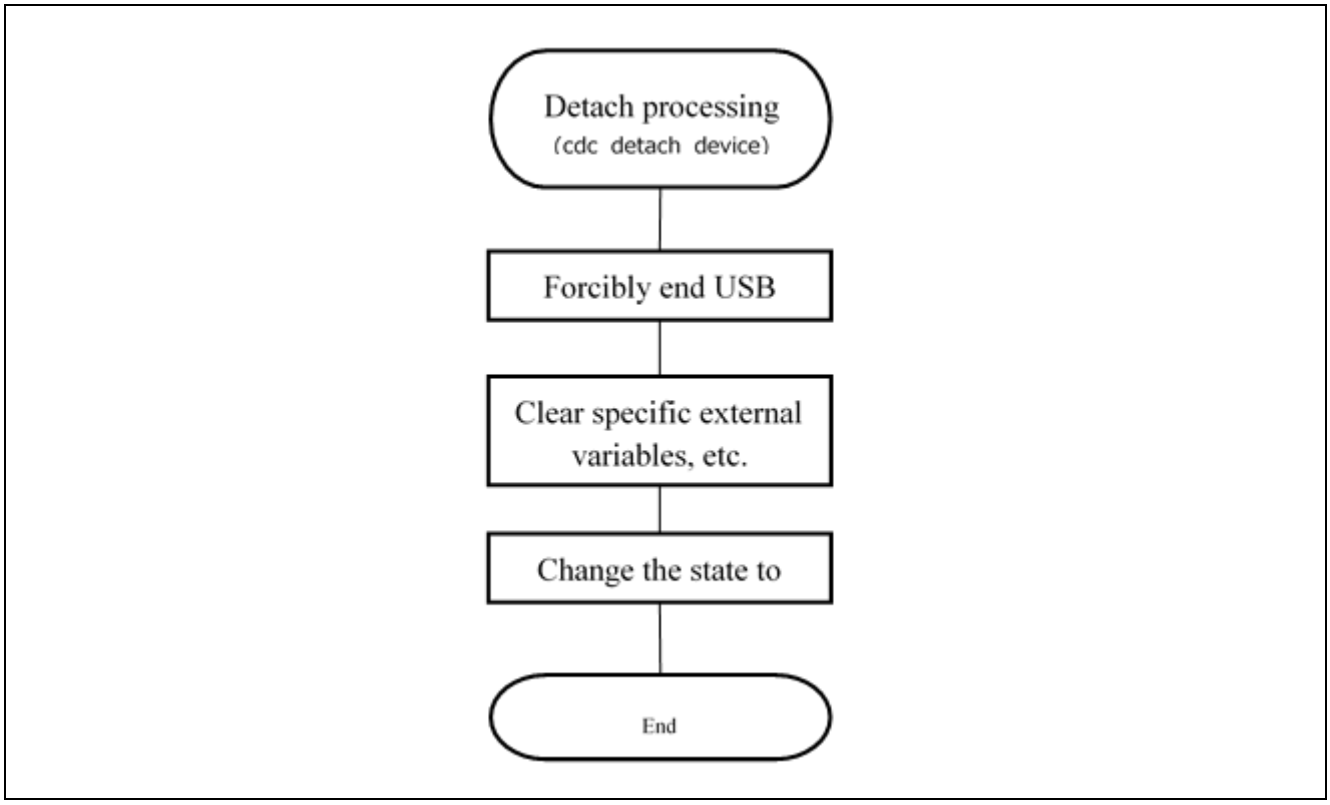


Figure 7-4 Detach Processing of the Sample Code

7.7 Tutorials

7.7.1 Operational Overview

The table below shows the main functions of sample application.

1. Initializes the USB.
2. Responses to CDC class requests.
3. Receives data from the USB host.
4. Sends the received data to the USB host.

Figure 7-5 shows a system block diagram.

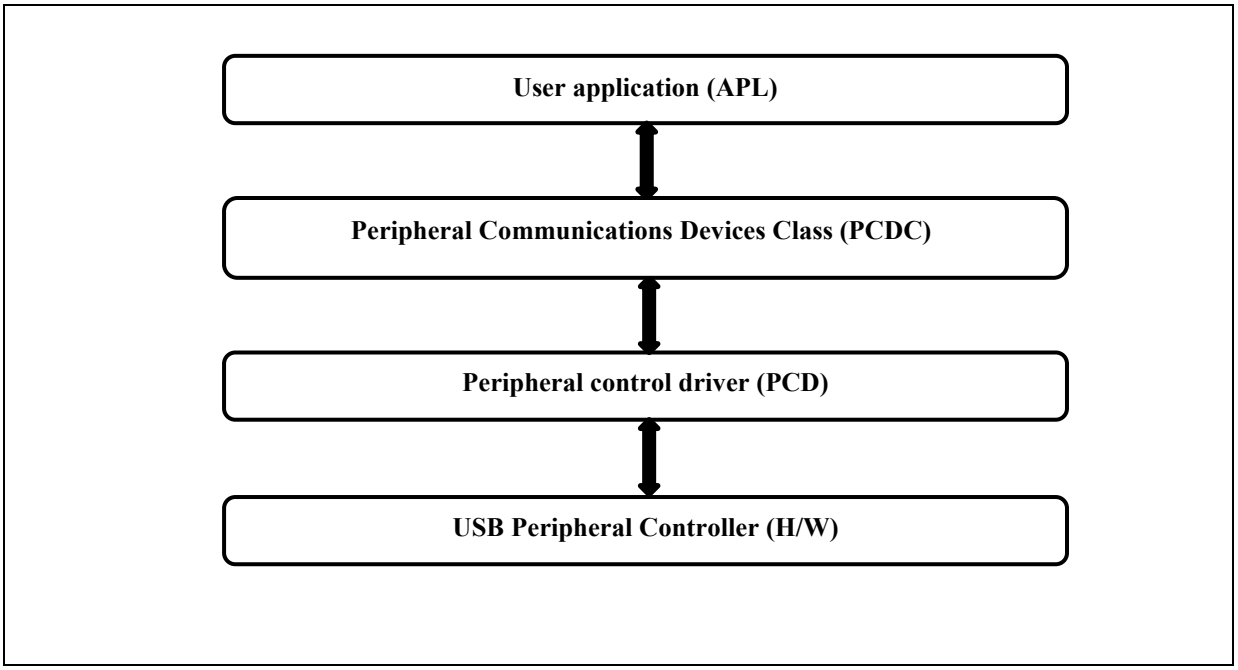


Figure 7-5 System Block Diagram

7.7.2 Preparations

Figure 7-6 shows an operation environment example.

Note: Install the device driver file under ../Source/Driver/usbf/pcdc/ utilities.

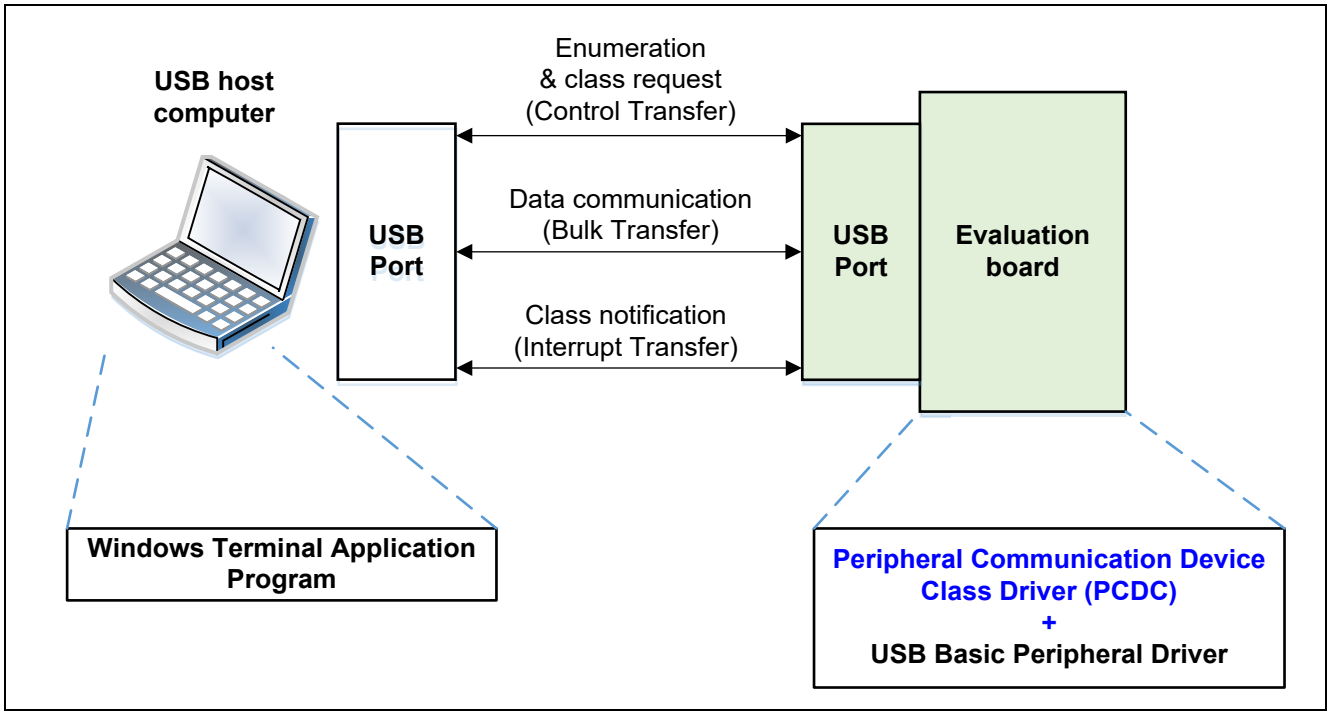


Figure 7-6 Connection Example

7.7.3 Terminal Software (Tera Term)

The settings of the terminal software in the host computer are as follows:

- Transfer rate: 115,200 bps
- Character length: 8 bits
- Stop bit length: 1 bit
- Parity function: None
- Flow control: Hardware control

7.7.4 Sample Program Settings

Shown below is an example of initial settings.

Initial setting example:

```
void usbf_main(void)
{
   /* See USB driver initial setting */
   pcdc_registration();
   /* See Starting the USB module */
   R_USB_Open();

   apl_init();
   /* See Main routine */
   pcdc_main();
}
```

USB driver initial setting

The USB driver setting registers class driver information for the PCD.
A PDCD must be created for the user system.
Information about the created PDCD must be registered in the USB_PCDREG_t structure by using API function R_usb_pstd_DriverRegistration (). For details about the registration procedure, see section 3.10.5, Registering the Peripheral Device Class Driver (PDCD).

Starting the USB module

By calling R_USB_Open(), the USB module can be set according to the initial setting sequence in the hardware manual, the USB interrupt handler can be registered, and the USB interrupt enable setting can be made.

Main routine

After the initial setting, the USB driver operates by calling the interrupt processing function (R_usb_pstd_poll) in the main routine of the application. Whether an interrupt is generated is checked by calling R_usb_pstd_poll() in the main routine. If an interrupt is generated, the main routine performs processing according to the generated interrupt.

```
void pcdc_main(void)
{
    while(1)
    {
        R_usb_pstd_poll();

        switch( cdc_dev_info.state )
        {
            case STATE_DATA_TRANSFER:
                cdc_data_transfer();
                break;
            case STATE_ATTACH:
                cdc_connect_wait();
                break;
            case STATE_DETACH:
                cdc_detach_device();
                break;
            default:
                break;
        }
    }
}
```

7.7.5 Sample Program Execution Example

(1) Connect the USB virtual COM port of EC-1 to the host computer and start the terminal software.

(2) The sample program waits for the data reception and sends an initial connection message at specified intervals.

(3) When data is transferred from the terminal software on the computer, “Echo Mode.” is displayed on the terminal software. After this display, loopback processing is performed.

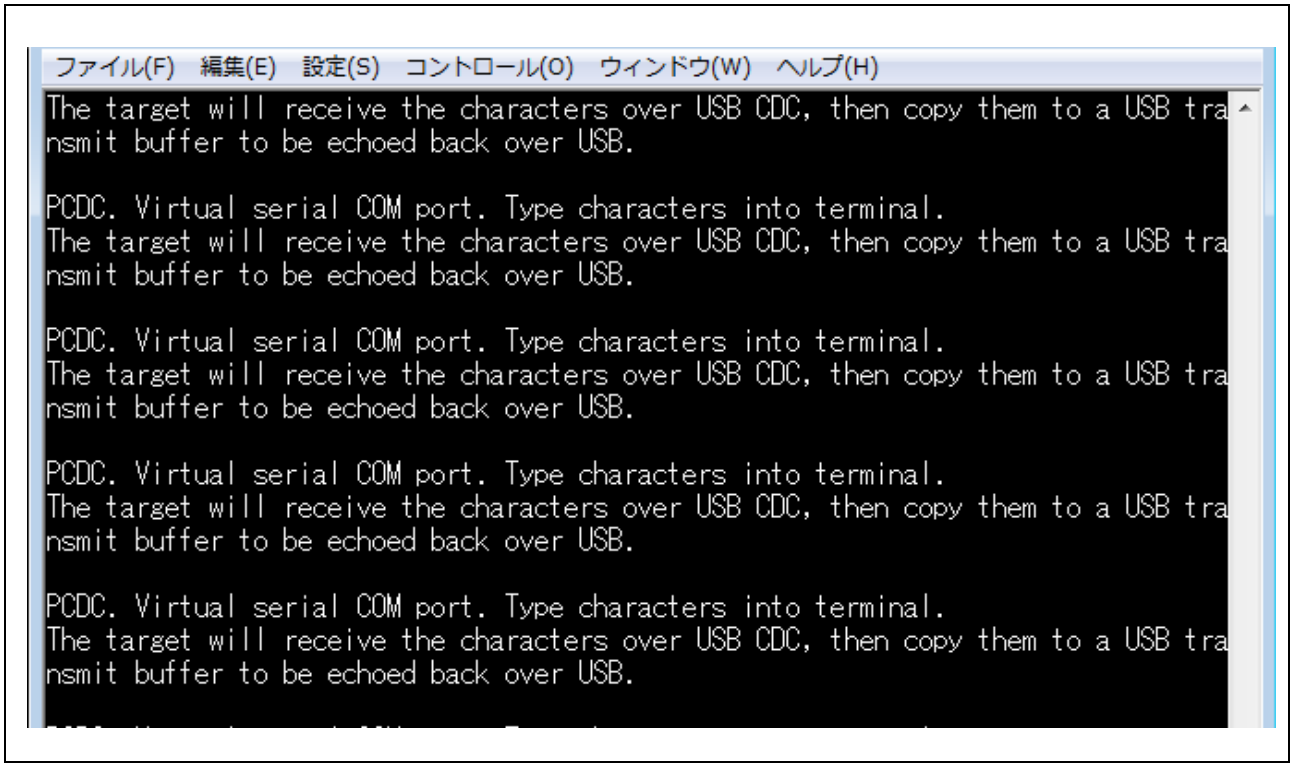


Figure 7-7 Waiting for Data Reception

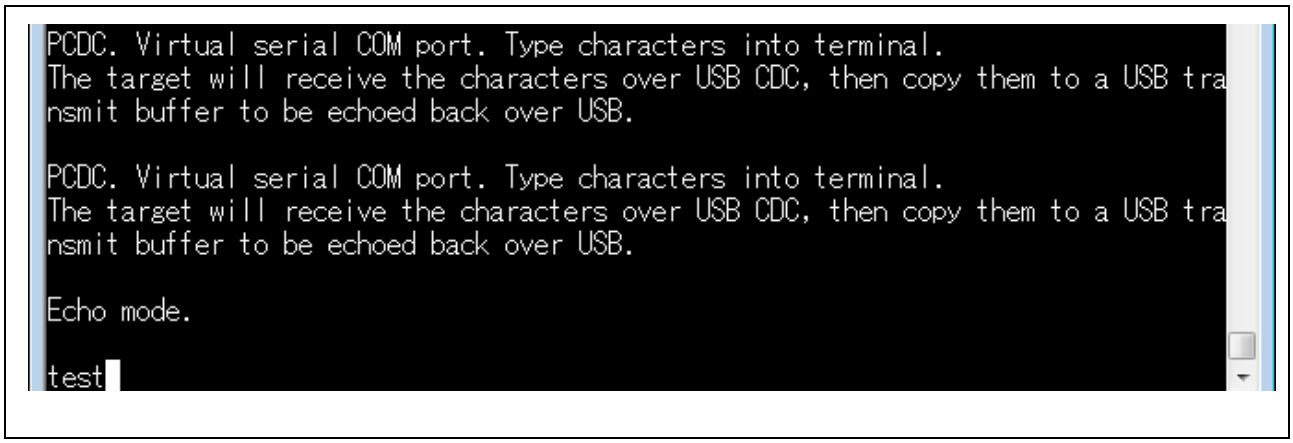


Figure 7-8 Loopback Processing

7.8 Peripheral Control Driver (PCD)

7.8.1 Basic Functions

The PCD is a program for controlling the hardware. It controls the hardware according to the USB interrupt source and notifies the PDCD of the result via the callback function.

It also analyzes the requests that the PDCD issues for hardware control.

The PCD functions are as follows:

1. Control transfer (ControlRead/ControlWrite/No-data Control)
2. Data transfer (Bulk/Interrupt) and result notification
3. Suspension of data transfer (all pipes)
4. Detection of a USB bus reset signal and notification of the reset handshake result
5. Suspend/resume detection
6. Detection of attach/detach triggered by a VBUS interrupt
7. Hardware control to the clock stopped state (low power mode) and restoration from it

7.8.2 Issuing a Request to the PCD

API functions are used to issue a hardware control request to the PCD and to perform data transfer. In response to a request from the upper layer, the PCD notifies the result by using a callback function.

The PCD does not have APIs that handle class/vendor requests.

7.8.3 USB Requests

The standard requests that the PCD handle are as follows:

GET_STATUS
GET_DESCRIPTOR
GET_CONFIGURATION
GET_INTERFACE
CLEAR_FEATURE
SET_FEATURE
SET_ADDRESS
SET_CONFIGURATION
SET_INTERFACE

The PCD issues a STALL response to any request other than above.

If the received USB request is a device class request or a vendor class request, create a function that identifies these requests and register them in the driver.

The following shows the processing of the control transfer stage when receiving a standard request.

ControlRead: Create data to be transmitted to the host and make a transmission request if receiving a correct request.

ControlWrite: Request data reception from the host if receiving a correct request.

NoDataControl: Make a status response if receiving a correct request.

ControlRead-Status: Make a status response.

ControlWrite-Status: Make a status response.

Control transfer completion: Notify the PDCD of a request from the host.

7.8.4 API Functions

The PDCD requests to control the hardware by using PCD API functions.

See Table 3-69 List of USB Function Driver Functions.

For details, see section 3.10, USB Function Control.

7.8.5 PCD Callback Functions

The tables below list the PCD callback functions.

Table 7-9 Callback Function for R_usb_pstd_TransferStart

Function call syntax	(*USB_UTR_CB_t)(USB_UTR_t *);
Arguments	USB_UTR_t* : Pointer to USB_UTR_t
Returned values	—
Description	Executed when data transfer is finished (transfer of data of the size specified by the application is finished or transfer is aborted by receiving a short packet, etc.). The length of data remained to be sent or received and the error count are updated.

Table 7-10 Callback Function for Control Transfer

Function call syntax	(*USB_CB_TRN_t)(USB_REQUEST_t * ,uint16_t)
Arguments	USB_REQUEST_t *: Request information uint16_t : Stage information
Returned values	—
Description	Executed when receiving control transfer except standard requests.

Table 7-11 Callback Function for USB State Change

Function call syntax	(*USB_CB_t)(uint16_t)
Arguments	uint16_t : See “Description” below.
Returned values	—
Description	This callback function is called by the PCD when the USB state changes. The following arguments are set in the callback function in each state.

State	Argument Overview
default	Operating device speed USB_HSCONNECT: High-speed USB_FSCONNECT: Full-speed USB_NOCONNECT: No connect
configured	Configuration number
detach	No argument used
suspend	Remote wakeup enable flag USB_TRUE: Enable USB_FALSE: Disable
resume	No argument used
interface	Alternate number

7.8.6 Interrupt Processing

When a USB interrupt is generated, the PCD identifies the source of the interrupt by the interrupt handler and stores the result in the USB_INT_t structure. The PCD does not control the hardware in the interrupt handler.

It controls the hardware when the user calls R_usb_pstd_poll() in the application.

Cautions

Global variable usb_gpstd_UsbInt declared in the USB_INT_t structure is managed by the PCD. If the user changes the value of this variable, the PCD does not work properly. Do not change the value of this variable.

7.9 Peripheral Device Class Driver (PDCD)

7.9.1 Registering the Peripheral Device Class Driver

A PDCD must be created for the user system.

Information about the created PDCD must be registered in the USB_PCDREG_t structure by using API function R_usb_pstd_DriverRegistration (). For details about the registration procedure, see section 3.10.5, Registering the Peripheral Device Class Driver (PDCD).

7.9.2 Peripheral Control Transfer

This section describes an example of peripheral control transfer program that uses the API functions provided by the PCD.

This example shows control transfer when receiving a class request.

Control transfer processing requires the following functions:

- Class request processing function (create one for each user system. For details about this function, see section 7.9.3, Class Request Processing Function.)
- API function for the data stage (R_usb_pstd_ControlRead() / R_usb_pstd_ControlWrite())
- API function for the status stage (R_usb_pstd_ControlEnd() / R_usb_pstd_ SetStall())

7.9.3 Class Request Processing Function

Upon receiving a class request, the PCD stores the content of the request in global variable usb_gpstd_ReqReg declared in the USB_PCDREG_t structure, and calls the class request processing function registered in member ctrltrans in the USB_PCDREG_t structure.

Request information and stage information in control transfer are set as arguments of this function. The class request processing function is called in the data stage and status stage.

An example of the class request processing function is shown on the next page.

<Example of the class request processing function>

```
void usb_pvendor_UsrCtrlTransFunction(USB_REQUEST_t *preq, uint16_t ctsq)
{
    if( preq->ReqTypeType == USB_CLASS )
    {
        switch( ctsq )
        {
        /* Idle or setup stage */
        case USB_CS_IDST: usb_ pvendor _ControlTrans0(preq); break;
        /* Control read data stage */
        case USB_CS_RDDS: usb_ pvendor _ControlTrans1(preq); break;
        /* Control write data stage */
        case USB_CS_WRDS: usb_ pvendor _ControlTrans2(preq); break;
        /* Control read nodata status stage */
        case USB_CS_WRND: usb_ pvendor _ControlTrans3(preq); break;
        /* Control read status stage */
        case USB_CS_RDSS: usb_ pvendor _ControlTrans4(preq); break;
        /* Control write status stage */
        case USB_CS_WRSS: usb_ pvendor _ControlTrans5(preq); break;

        /* Control sequence error */
        case USB_CS_SQER: R_usb_pstd_ControlEnd((uint16_t)USB_DATA_ERR); break;
        /* Illegal */
        default: R_usb_pstd_ControlEnd((uint16_t)USB_DATA_ERR); break;
        }
    }
    else
    {
        R_usb_pstd_SetPipeStall ((uint16_t)USB_PIPE0);
    }
}
```

1. Data stage processing

If the received request is supported, perform data transfer to the host using API function R_usb_pstd_ControlRead()/R_usb_pstd_ControlWrite().

For an unsupported request, call API function R_usb_pstd_SetStall to return STALL to the host.

The class request processing function above calls the following functions:

(a) usb_ pvendor _ControlTrans1()

(b) usb_ pvendor _ControlTrans2()

2. Status stage processing

If there is no problem with the setup or data stage, call API function R_usb_pstd_ControlEnd() with USB_DATA_END specified as an argument.

When receiving a parameter that is not supported by the setup or data stage, call API function R_usb_pstd_SetStall() to return STALL.

The class request processing function above calls the following functions:

(a) usb_ pvendor _ControlTrans3()

(b) usb_ pvendor _ControlTrans4()

(c) usb_ pvendor _ControlTrans5()

7.10 Peripheral Communication Device Driver (PCDC)

7.10.1 Basic Functions

The PCDC conforms with the abstract control model subclass of the Emulation Device Class Specifications.

The main functions of the PCDC are as follows:

1. Respond to a function inquiry from the USB host
2. Respond to a class request from the USB host
3. Data communication with the USB host
4. Report a serial communication error to the USB host

7.10.2 Overview of the Abstract Control Model

The Abstract Control Model subclass bridges the gap between legacy modem devices and USB devices, enabling the use of application programs designed for older modems.

The following sections describe the class requests and class notifications that the PCDC supports.

7.10.3 Class Requests (Notification from the Host to the Device)

This section describes the class requests that the PCDC supports.

Table 7-12 CDC Class Requests

Request	Code	Description	Support
SendEncapsulatedCommand	0x00	Sends the AT command, etc. defined in the protocol.	No
GetEncapsulatedResponse	0x01	Requests a response to the command sent in SendEncapsulatedCommand.	No
SetCommFeature	0x02	Enables or disables the 2-byte code unique to the device and country setting.	No
GetCommFeature	0x03	Gets the enable or disable setting of the 2-byte code unique to the device and country setting.	No
ClearCommFeature	0x04	Sets the enable or disable setting of the 2-byte code unique to the device and country setting to the default state.	No
SetLineCoding	0x20	Makes communication line settings. (Transfer rate, data length, parity bit and stop bit length)	Yes
GetLineCoding	0x21	Gets the communication line settings.	Yes
SetControlLineState	0x22	Makes settings for communication line control signals RTS and DTR.	Yes
SendBreak	0x23	Sends a break signal.	No

For details about the abstract control model requests, refer to Table11: Requests - Abstract Control Model in the USB Communications Class Subclass Specification for PSTN Devices, Revision 1.2.

7.10.4 Data Format of Class Requests

This section describes the data format of the class requests that the PCDC supports.

(1) SetLineCoding

This class request is sent from the host to the device to make UART line settings.

The table below shows the data format of SetLineCoding.

Table 7-13 SetLineCoding Format

bmRequestType_t	bReques	wValue	wIndex	wLength	Data
0x21	SET_LINE_CODING (0x20)	0x00	0x00	0x07	See Table 7-14.

Table 7-14 Line Coding Structure Format

Offset	Field	Size	Value	Description
0	DwDTERate	4	Number	Transfer rate (bps) of the data terminal
4	BcharFormat	1	Number	Stop bit length 0: 1 stop bit 1: 1.5 stop bits 2: 2 stop bits
5	BparityType	1	Number	Parity 0: None 1: Odd 2: Even
6	BdataBits	1	Number	Data bits (5, 6, 7 and 8)

(2) GetLineCoding

This class request is sent from the host to the device to request UART line setting.

The table below shows the data format of GetLineCoding.

Table 7-15 GetLineCoding Format

bmRequestType_t	bReques	wValue	wIndex	wLength	Data
0xA1	GET_LINE_CODING (0x21)	0x00	0x00	0x07	See Table 7-14.

(3) SetControlLineState

This class request is sent from the host to the device to set UART flow control signals. The PCDC does not support the RTS or DTR control.

The table below shows the data format of SET_CONTROL_LINE_STATE.

Table 7-16 SET_CONTROL_LINE_STATE Format

bmRequestType_t	bReques	wValue	wIndex	wLength	Data
0x21	SET_CONTROL_LINE_ STATE (0x22)	See Table 7-17.	0x00	0x00	None

Table 7-17 Control Signal Bitmap Format

Bit Position	Description
D15 to D2	Reserved (0)
D1	Controls the transmission function of the DCE. 0: RTS OFF 1: RTS ON
D0	Notifies whether the DTC is ready. 0: DTR not present 1: DTR present

7.10.5 Class Notification (Notification from the Device to the Host)

The table below shows which class notifications the PCDC supports.

Table 7-18 CDC Class Notifications

Notification	Code	Description	Support
NETWORK_CONNECTION	0x00	Notifies the network connection state.	No
RESPONSE_AVAILABLE	0x01	Response to GET_ENCAPSLATED_RESPONSE	No
SERIAL_STATE	0x20	Notifies the state of the serial line.	Yes

(1) SerialState

This class notification notifies the host of the state when detecting a state change of the UART.

The PCDC supports the detection of overrun errors, parity errors and framing errors. The state is notified when an error is detected. Even if more than one error is detected, another state notification is not sent.

The table below shows the data format of SerialState.

Table 7-19 SerialState Format

bmRequestType_t	bReques	wValue	wIndex	wLength	Data
0xA1	SERIAL_STATE (0x20)	0x00	0x00	0x02	See Table 7-20.

Table 7-20 UART State bitmap Format

Bit Position	Field	Description
D15 to D7		Reserved
D6	bOverRun	An overrun was detected.
D5	bParity	A parity error was detected.
D4	bFraming	A framing error was detected.
D3	bRingSignal	A ring signal was sensed.
D2	bBreak	A break signal was detected.
D1	bTxCarrier	Data Set Ready: The line is connected and ready for communication.
D0	bRxCarrier	Data Carrier Detect: The carrier is detected on the line.

7.10.6 Virtual COM Port of the Computer

A computer running on Windows OS can use a CDC device as the virtual COM port.

If you connect an evaluation board with the PCDC to a Windows OS computer, CDC class requests GetLineCoding and SetControlLineState are made after an enumeration setting, and the evaluation board is registered as a virtual COM device. After registered as a virtual COM device in the Windows device manager, the evaluation board can communicate with the computer by using the terminal application such as Hyper Terminal installed in the Windows OS computer as standard.

By making serial port settings in the terminal application, communication line settings can be made by class request SetLineCoding. The data (or file) input from the window of the terminal application is transferred to the evaluation board. The data transferred from the evaluation board is output to the window of the terminal application.

If the last received data is less than the maximum packet size, some terminal applications do not display the received data on the terminal, anticipating more data. In this case, the received data is displayed on the terminal by receiving data that is less than the maximum packet size.

7.10.7 APIs

All API calls and the interface definitions supporting these API calls are described in r_usb_pcdc_if.h.

Make configuration option settings for this module with r_usb_pcdc_config.h.

Table 7-21 lists the option names and setting values.

Table 7-21 PCDC Configuration Options

Definition Name	Default	Description
USB_PCDC_USE_PIPE_IN	USB_PIPE1	Bulk IN transfer pipe number
USB_PCDC_USE_PIPE_OUT	USB_PIPE2	Bulk OUT transfer pipe number
USB_PCDC_USE_PIPE_STATUS	USB_PIPE6	Interrupt IN transfer pipe number

Table 7-22 lists the PCDC APIs.

Table 7-22 PCDC APIs

Function Name	Function Overview
R_usb_pcdc_SendData	USB transmission processing
R_usb_pcdc_ReceiveData	USB reception processing
R_usb_pcdc_SerialStateNotification	Class notification “SerialState”
R_usb_pcdc_ctrltrans	Control transfer for CDC

7.11 Pipe Information Table

To perform data communications using pipes 1 to 9, the PDCD must maintain an information table to define pipe settings. The pipe information table includes the values to be set in the register that are required for data transfer and settings for the transfer method (CPU transfer or DMA transfer) for the FIFO ports.

A configuration example of the pipe information table is shown below.

```
uint16_t usb_gvendor_smpl_eptbl[] =
{
USB_PIPE1,                                  ← Pipe definition item 1
USB_BULK | USB_BFREOFF | USB_DBLBON |
USB_CNTMDON | USB_SHTNAKON | USB_EP1 |
USB_DIR_P_IN,                               ← Pipe definition item 2
USB_BUF_SIZE(1024) | USB_BUF_NUMB(8),       ← Pipe definition item 3
512,                                        ← Pipe definition item 4
USB_IFISOFF | USB_IITV_TIME(0u),            ← Pipe definition item 5
USB_CUSE,                                   ← Pipe definition item 6
:
:
USB_PDTBEND,
}
```

The pipe information table consists of the following six items (uint16_t×6).

1. Pipe window selection register (address 0x64)
2. Pipe configuration register (address 0x68)
3. Pipe buffer specification register (address 0x6A)
4. Pipe maximum packet size register (address 0x6C)
5. Pipe cycle control register (address 0x6E)
6. FIFO port usage

(1) Pipe definition item 1

Specify a value to be set in the pipe window selection register.

Pipe selection: Specify the pipe to be used (USB_PIPE1 to USB_PIPE9).

Restrictions

-

(2) Pipe definition item 2

Specify values to be set in the pipe configuration register.

Transfer type: Specify either USB_BULK/USB_INT or USB_ISO.

BRDY interrupt operation specification: Specify USB_BFREOFF.

Double-buffer mode: Specify either USB_DBLBON or USB_DBLBOFF.

Continuous transfer mode: Specify either USB_CNTMDON or USB_CNTMDOFF.

SHTNAK operation specification: Specify either USB_SHTNAKON or USB_SHTNAKOFF.

Transfer direction: Specify either USB_DIR_P_OUT or USB_DIR_P_IN.

Endpoint number: Specify an endpoint number (EP1 to EP15) for the pipe.

Restrictions

1. For the transfer type, the settable value differs depending on the selected pipe. For details, refer to the User's Manual.
2. If the transfer direction is set to reception (USB_DIR_P_OUT), specify USB_SHTNAKON for the SHTNAK operation.

(3) Pipe definition item 3

Specify values to be set in the pipe buffer specification register.

Buffer size: Specify the pipe buffer size in units of 64 bytes.

Buffer number: Specify the first block number of the pipe buffer.

Restrictions

- Specify values so that the buffer areas used do not overlap.
- If USB_DBLBON is set for double-buffer mode in pipe definition item 2, twice as large as the set buffer area is necessary.

(4) Pipe definition item 4

Specify a value to be set in the pipe maximum packet size register.

Maximum packet size: Specify the maximum packet size of the pipe.

Restrictions

-

(5) Pipe definition item 5

Specify values to be set in the pipe cycle control register.

ISO IN buffer flush: Specify USB_IFISOFF.

Interval: Specify a value (0 to 7) for the transfer interval timing.

Restrictions

- If the transfer type is not ISO IN, specify USB_IFISOFF for the ISO IN buffer flush.
- If USB_PIPE3 to USB_PIPE5 is selected, specify value 0 for the interval.

(6) Pipe definition item 6

Specify the FIFO port and access method that the pipe uses. The following values can be set:

USB_CUSE: The CPU accesses CFIFO.

USB_D0DMA: The DMA accesses D0FIFO (cycle steal mode).

USB_D0DMA_C: The DMA accesses D0FIFOBn (32-byte continuous access mode).

USB_D1DMA_C: The DMA accesses D1FIFOBn (32-byte continuous access mode).

Restrictions

- The receiving pipes use the transaction counter.
- There is no sample function for USB_D1DMA.
- For DMA transfer, the same port access method cannot be specified for different pipes.

Example: When USB_D0DMA_C is specified for USB_PIPE1, USB_D0DMA_C or USB_D0DMA cannot be specified for USB_PIPE2.

(7) Other restrictions

- Communication must be synchronized in transfer units in the device class.
- Write USB_PDTBLEND at the end of the pipe information table.

7.12 User Definition Information File

The PCD functions can be configured by rewriting the user definition information file (r_usb_basic_config.h).

Change the following items according to the system:

(1) DMA transfer setting

Specify whether or not DMA transfer is used.

#define USB_DMA_PP USB_DMA_USE_PP: DMA transfer used

#define USB_DMA_PP USB_DMA_NOT_USE_PP: DMA transfer not used

[Initial setting]

#define USB_DMA_PP USB_DMA_USE_PP

(2) Low power mode setting

Specify whether or not low power mode is used.

#define USB_CPU_LPW_PP USB_LPWR_USE_PP: Low power mode used

#define USB_CPU_LPW_PP USB_LPWR_NOT_USE_PP: Low power mode not used

[Initial setting]

#define USB_CPU_LPW_PP USB_LPWR_USE_PP

(3) Debugging information output function setting

Specify whether or not debugging information is displayed.

#define USB_DEBUG_OUTPUT_PP USB_DEBUG_ON_PP: Debugging information displayed

#define USB_DEBUG_OUTPUT_PP USB_DEBUG_OFF_PP: Debugging information not displayed

[Initial setting]

#define USB_DEBUG_OUTPUT_PP USB_DEBUG_OFF_PP

[Note]

This macro outputs debugging information on to the UART or the display device. It needs a serial driver and a driver for the display device.

7.13 Data Transfer

7.13.1 Data Transfer Requests

Data transfer can be requested by passing the USB_UTR_t structure with transfer information set as the argument of the R_usb_pstd_TransferStart() API function.

For details about the USB_UTR_t structure, see Table 3-21 USB_UTR_t Structure.

7.13.2 Notification of Transfer Result

When data transfer is completed, the PCD notifies the PDCD of the end of data transfer and sequence toggle bit information by using the callback function registered at the time of data transfer request. The transfer results are stored in member status of the USB_UTR_t structure.

The following are communication results:

USB_DATA_NONE: When data transmission was ended normally

USB_DATA_OK: When data reception was ended normally

USB_DATA_SHT: When the data length was less than the specified length although data reception was ended normally

USB_DATA_OVR: When the received data exceeded the specified size

USB_DATA_ERR: When no response or overrun/underrun error was detected

USB_DATA_STALL: When a STALL response or MaxPacketSiz error was detected

USB_DATA_STOP: When data transfer was ended forcibly

7.13.3 Notes on Data Reception

- If a short packet is received, the data length remained to be received is stored in tranlen of the USB_UTR_t structure and the transfer is ended.

  If the received data is larger than the user buffer, data of the user buffer size is read from the FIFO buffer and transfer is ended.

  If the transfer size cannot be secured in the user buffer area, other data areas might be cleared.

- When DMA is used, the buffer that stores the received data needs an area an integral multiple of the maximum packet size of the pipe used for transfer.

  The table below shows the buffer sizes required for the received data sizes when the maximum packet size is 64 bytes.

Table 7-23 Buffer Size Examples (Max Packet Size Is 64 Bytes)

Received Data Size	Buffer Size [Bytes]
64 bytes or less	64 (maximum packet size × 1)
65 bytes or more and 128 bytes or less	128 (maximum packet size × 2)
…	…
449 bytes or more and 512 bytes or less	512 (maximum packet size × 8)
…	…

7.13.4 **Data Transfer Example**

The following is a data transfer example:

(1) Set transfer status in the following members of the USB_UTR_t structure by using the APL or PDCD.

keyword	:	Pipe number
tranadr	:	Data buffer
tranlen	:	Transfer size
complete	:	Callback function executed when data transfer is completed

(2) Call R_usb_pstd_TransferStart() to request data transfer.

(3) After data transfer is completed, the callback function set in complete in (1) above is called to notify the transfer result. (See section 7.13.2, Notification of Transfer Result.)

7.14 DMA Transfer

7.14.1 Basic Specifications

The PCD can perform DMA data transfer between the user buffer and the FIFO buffer by using the DMA controller.
For DMA transfer, 32-byte continuous access mode or cycle steal mode can be selected. For how to set each access mode, see section 7.11, Pipe Information Table.
Table 7-24 overviews the 32-byte continuous access mode.
Table 7-25 overviews the cycle steal mode.

Table 7-24 32-Byte Continuous Access Mode

USB module settings		
FIFO port used	D0FIFOBn port/D0FIFO port	D1FIFOBn port/D1FIFO port
Access bit width	Transmission: 32 bits/8 bits Reception: 32 bits	Transmission: 32 bits/8 bits Reception: 32 bits
Interrupts	Transmission: D0FIFO interrupt/BEMP interrupt Reception: BRDY interrupt	Transmission: D1FIFO interrupt/BEMP interrupt
Data transfer size	Transmission: FIFO size × N + fraction Reception: (FIFO size/32) × 32	Transmission: FIFO size × N + fraction Reception: (FIFO size/32) × 32
DMA module settings		
Channel	DMAC0_0	DMAC0_1
Transfer data unit	Transmission: 256 bits/8 bits Reception: 256 bits	Transmission: 256 bits/8 bits Reception: 256 bits
Transfer mode	Single transfer	Single transfer
DMA mode	Register mode	Register mode
Interval function	Not used	Not used
Skip function	Not used	Not used
Suspend function	Not used	Not used
Buffer read function	Not used	Not used
Interrupts	Transmission: Transfer completion interrupt Reception: -	Transmission: Transfer completion interrupt

Table 7-25 Cycle Steal Mode

USB module settings		
FIFO port used	D0FIFO port	-
Access bit width	Transmission: 32 bits/8 bits Reception: 32 bits	-
Interrupts	Transmission: D0FIFO interrupt/BEMP interrupt Reception: BRDY interrupt	-
Data transfer size	Transmission: FIFO size × N + fraction Reception: (FIFO size/4) × 4	-
DMA module settings		
Channel	DMAC0_0	-
Transfer data unit	Transmission: 32 bits/8 bits Reception: 32 bits	-
Transfer mode	Single transfer	-
DMA mode	Register mode	-
Interval function	Not used	-
Skip function	Not used	-
Suspend function	Not used	-
Buffer read function	Not used	-
Interrupts	Transmission: Transfer completion interrupt Reception: -	-

8. WDTA Sample Software

8.1 Overview

This section describes the sample program that uses the WDTA driver for controlling channel 0 of the WDT (WDT0) of the MCU mounted on the evaluation board.

The features of the WDTA sample program are as follows.

- It refreshes the watchdog timer (WDTA) periodically (every 223.7 ms) based on the compare match timer (CMT) after the watchdog timer starts operating.
- When a software wait is generated by an external interrupt (IRQ2), the refresh operation is stopped and a reset is generated. LED2 lights up after a reset.

Target Devices

EC-1

When applying the sample program covered in this application note to another microcontroller, modify the program according to the specifications for the target microcontroller and extensively evaluate the modified program.

8.2 Constants

Table 8-1 lists the constants used in this sample code.

Table 8-1 Constants Used in the Sample Program

Constant Name	Setting Value	Description
CMT0_CLOCK_PCLKD_512	3	Count clock = PCLKD/512
CMT0_CMI0_ENABLE	1	Enables CMI0 interrupts.
CMT0_INTERVAL_TIME	0x7FFF	Sets interval time to 223.7 ms.
CPG_CMT0_START	1	Starts CMT0 count.

8.3 Functions

A list of the functions is shown in Table 8-2 List of Functions.

Table 8-2 List of Functions

Function Name	Description	Scope	Definition File
main	Main processing	local	main.c
port_init	Initializes the port settings	local	main.c
ecm_init	Initializes the ECM settings	local	main.c
icu_init	Sets interrupts	local	main.c
cmt_init	Initializes the CMT module	local	main.c
wdt0_init	Initializes WDT0	local	main.c
soft_wait	Software wait	local	main.c
R_IRQ2_isr	IRQ2 interrupt (IRQ pin interrupt 2) processing	local	main.c
R_IRQ21_isr	IRQ21 interrupt (compare match timer (CMI0)) processing	local	main.c

8.4 Details of the Functions

8.4.1 main

(1) **Synopsis**

Main processing

(2) **C language format**

void main (void);

(3) **Parameters**

None

(4) **Description**

This function is the main processing of the sample program:

- Initialization of the port settings.
- Initialization of the ECM.
- Initialization of the CMT.
- Initialization of the ICU.
- Initialization of the CMT.

It makes the initial settings above to start CMT0. Then it repeats turning LED1 on and off in the main loop.

(5) **Returned values**

None

8.4.2 port_init

(1) Synopsis

Initializes the port settings.

(2) C language format

void port_init (void);

(3) Parameters

None

(4) Description

This function initializes the port settings.

(5) Returned values

None

8.4.3 ecm_init

(1) Synopsis

Initializes the ECM settings.

(2) C language format

void ecm_init (void);

(3) Parameters

None

(4) Description

This function initializes the ECM settings.

(5) Returned values

None

8.4.4 icu_init

(1) **Synopsis**

Sets interrupts.

(2) **C language format**

void icu_init (void);

(3) **Parameters**

None

(4) **Description**

This function enables interrupt processing.

(5) **Returned values**

None

8.4.5 cmt_init

(1) **Synopsis**

Initializes the CMT module.

(2) **C language format**

void cmt_init (void);

(3) **Parameters**

None

(4) **Description**

This function initializes channel 0 of the CMT.

(5) **Returned values**

None

8.4.6 wdt0_init

(1) **Synopsis**

Initializes WDT0.

(2) **C language format**

void wdt0_init (void);

(3) **Parameters**

None

(4) **Description**

This function initializes WDT0 and starts counting by WDT0.

(5) **Returned values**

None

8.4.7 soft_wait

(6) **Synopsis**

Software wait processing

(1) **C language format**

void soft_wait(void);

(3) **Parameters**

None

(4) **Description**

This function handles software wait processing by using the NOP command.

(5) **Returned values**

None

8.4.8 R_IRQ2_isr

(1) **Synopsis**

IRQ2 interrupt (IRQ pin interrupt 2) processing

(2) **C language format**

void R_IRQ2_isr (void);

(3) **Parameters**

None

(4) **Description**

This function handles software wait processing (about 3 seconds). Within this time, the watchdog timer will underflow and the ECM will be reset.

After release from the reset state, LED2 is lit up in response to judgment that there has been a reset.

(5) **Returned values**

None

8.4.9 R_IRQ21_isr

(1) **Synopsis**

IRQ21 interrupt (compare match timer (CMI0)) processing

(2) **C language format**

void R_IRQ21_isr (void);

(3) **Parameters**

None

(4) **Description**

This function refreshes the watchdog timer.

(5) **Returned values**

None

8.5 Flowcharts

8.5.1 Main Processing

The figure below is a flowchart of the main processing of the sample code.

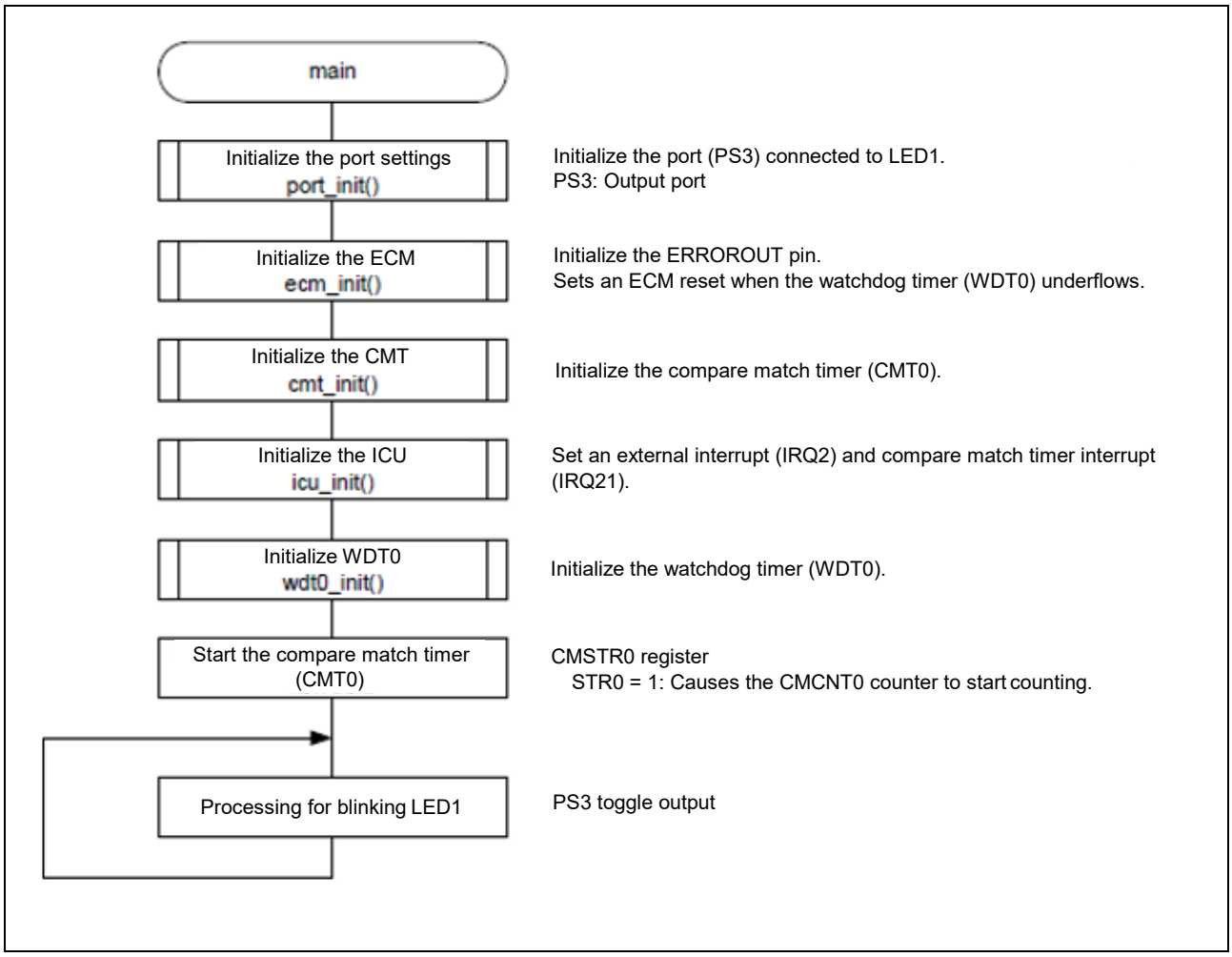


Figure 8-1 Main Processing of the Sample Code

8.5.2 Initialization of WDT0

The figure below is a flowchart of processing for initialization of WDT0.

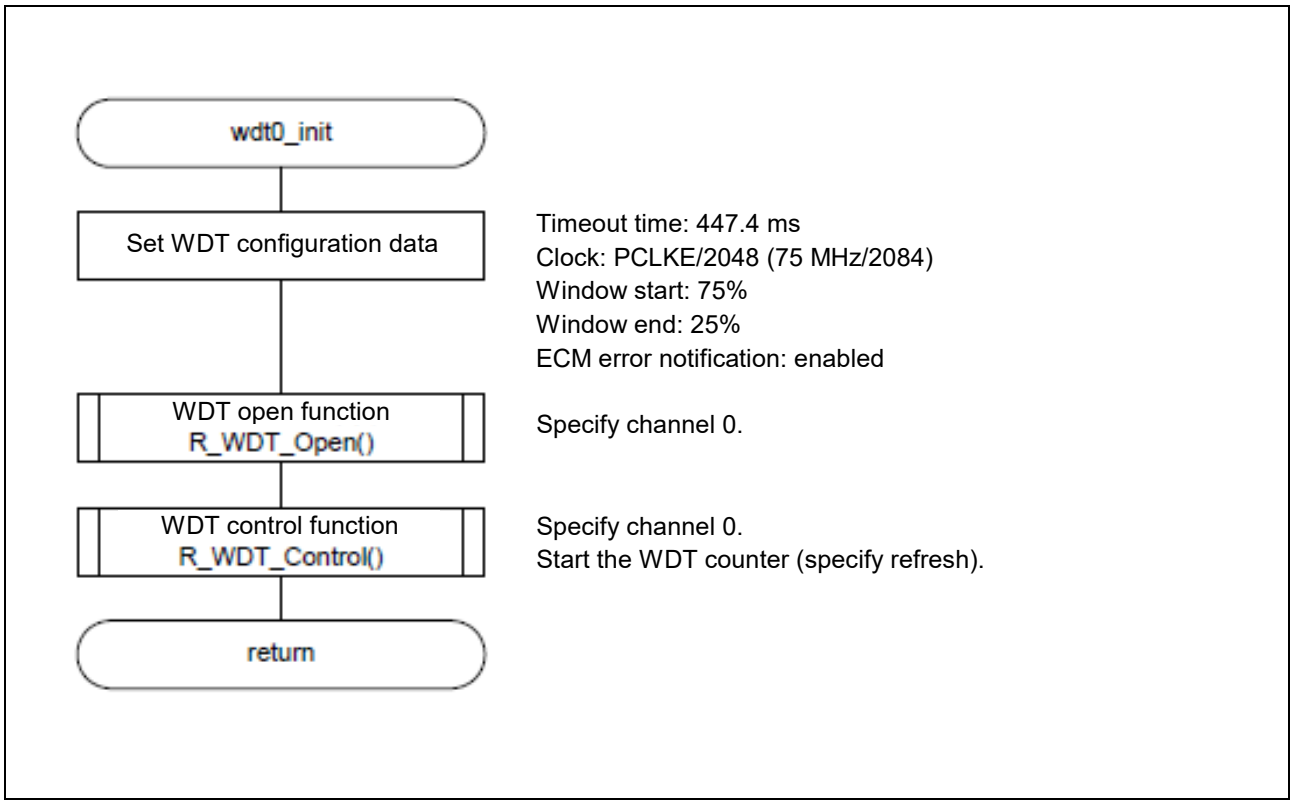


Figure 8-2 Initialization of wdt0

8.5.3 WDT Open Function

The figure below is a flowchart of the WDT open function.

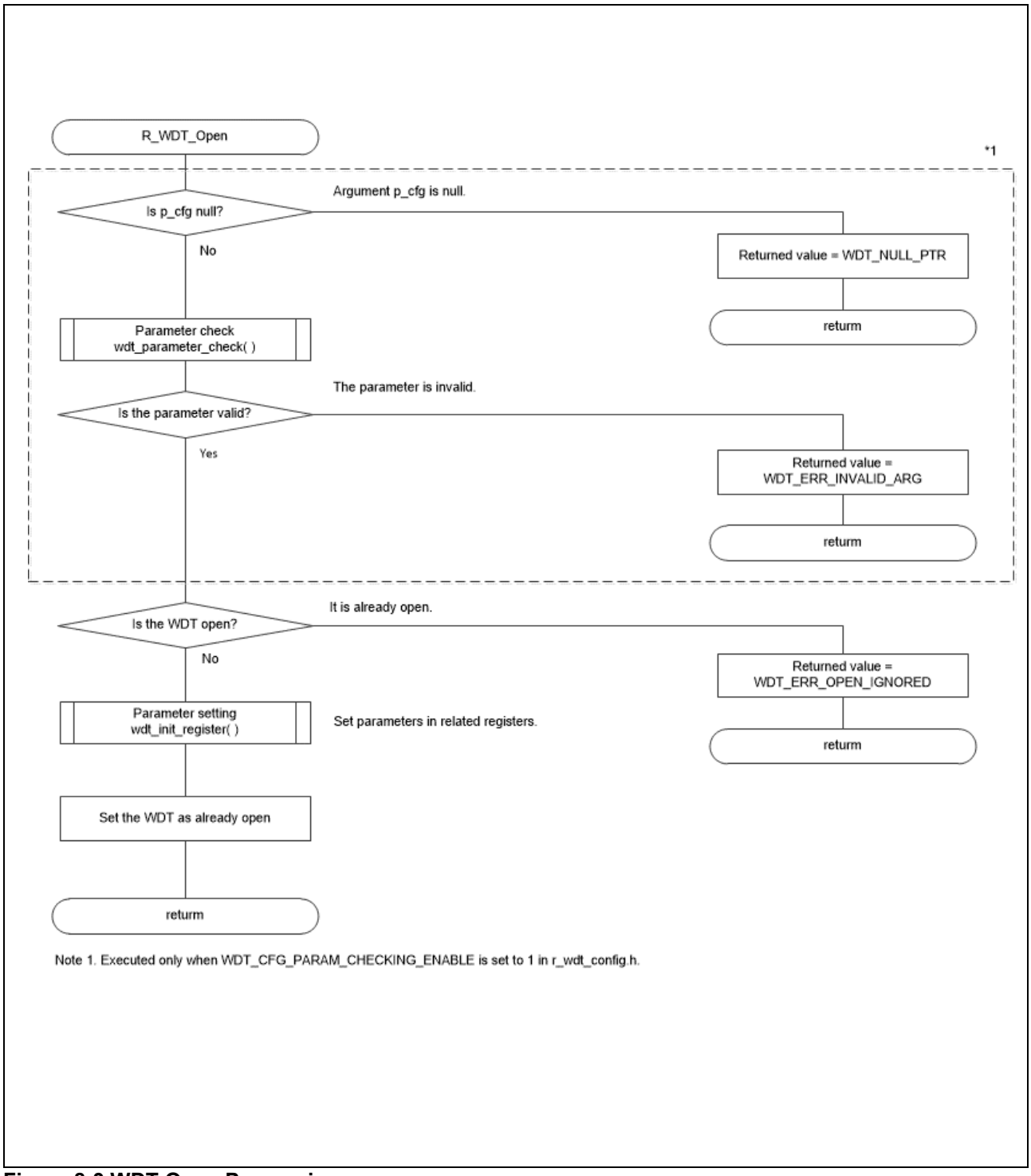


Figure 8-3 WDT Open Processing

8.5.4 WDT Control Function

The figure below is a flowchart of the WDT control function.

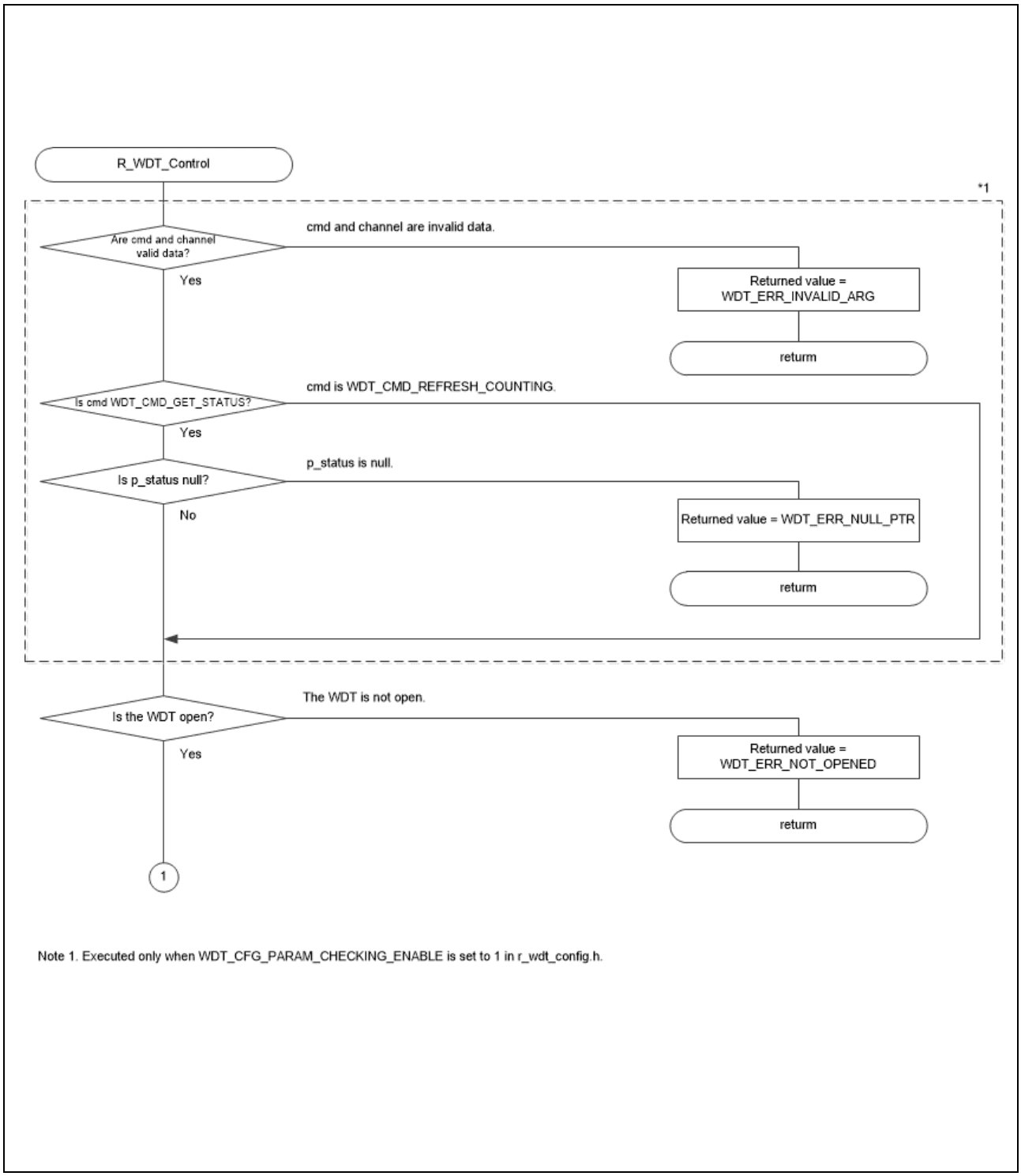


Figure 8-4 WDT Control Processing

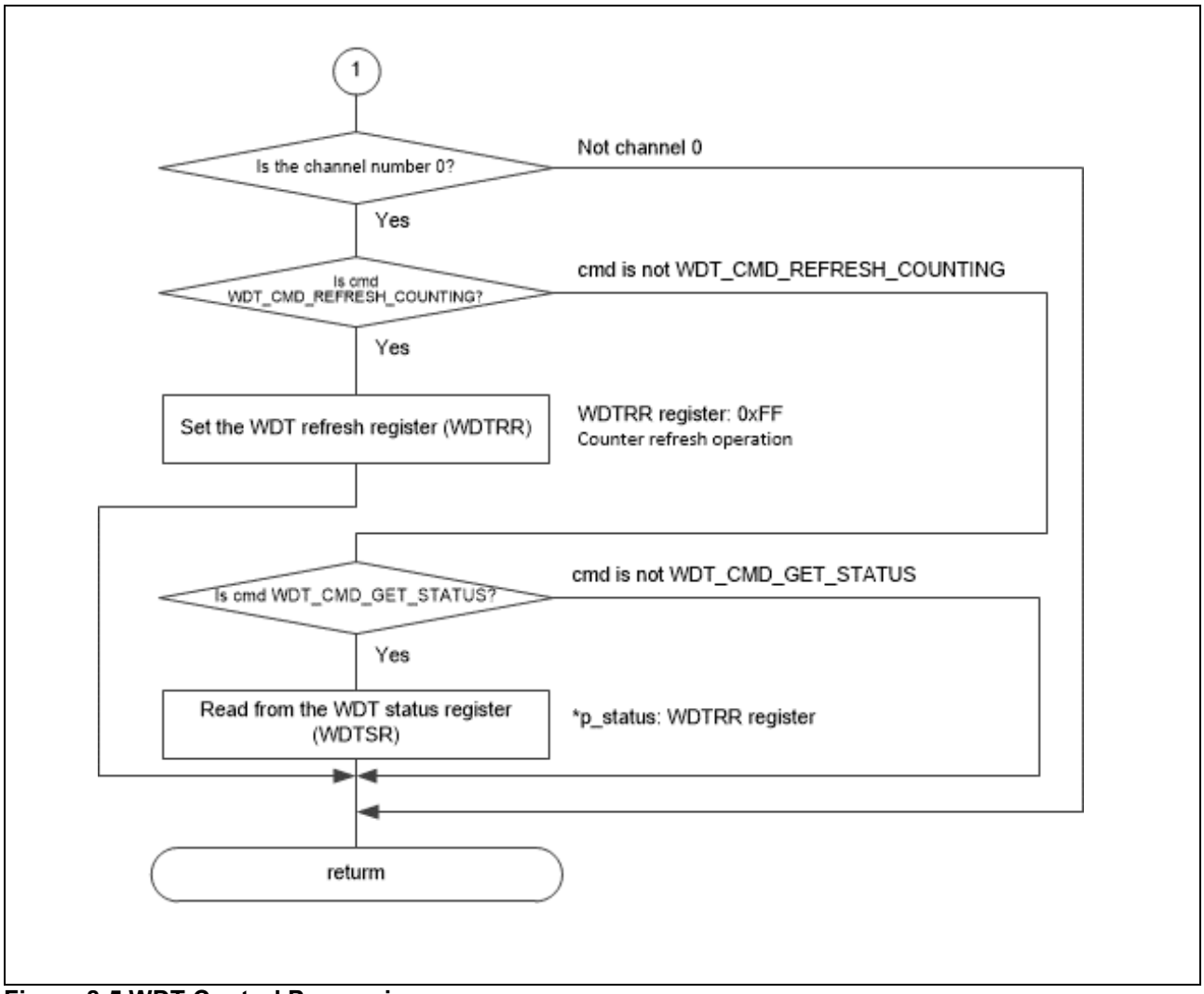


Figure 8-5 WDT Control Processing

8.5.5 IRQ2 Interrupt (IRQ Pin Interrupt 2) Processing

The figure below is a flowchart of the IRQ2 interrupt (IRQ pin interrupt 2) processing.

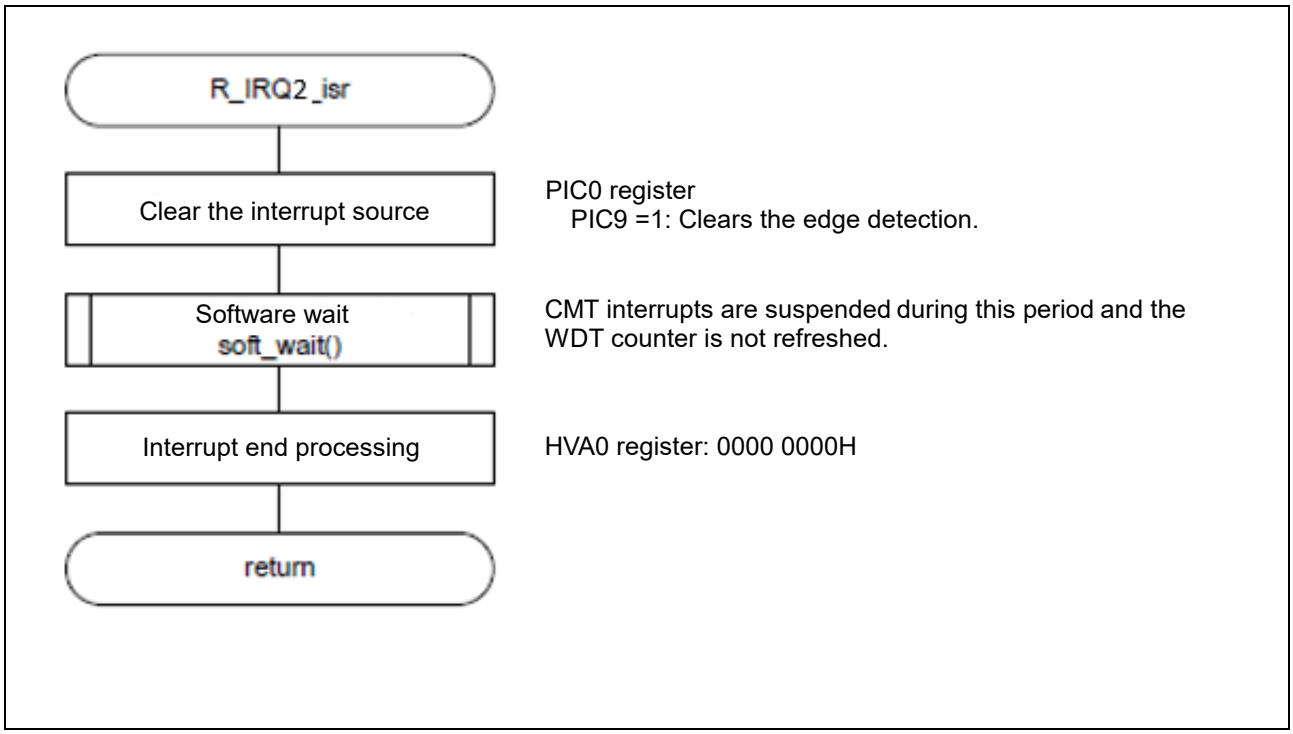


Figure 8-6 IRQ2 Interrupt (IRQ Pin Interrupt 2) Processing

8.5.6 IRQ21 Interrupt (Compare Match Timer 0 Interrupt) Processing

The figure below is a flowchart of the IRQ21 interrupt (compare match timer channel 0 interrupt) processing.

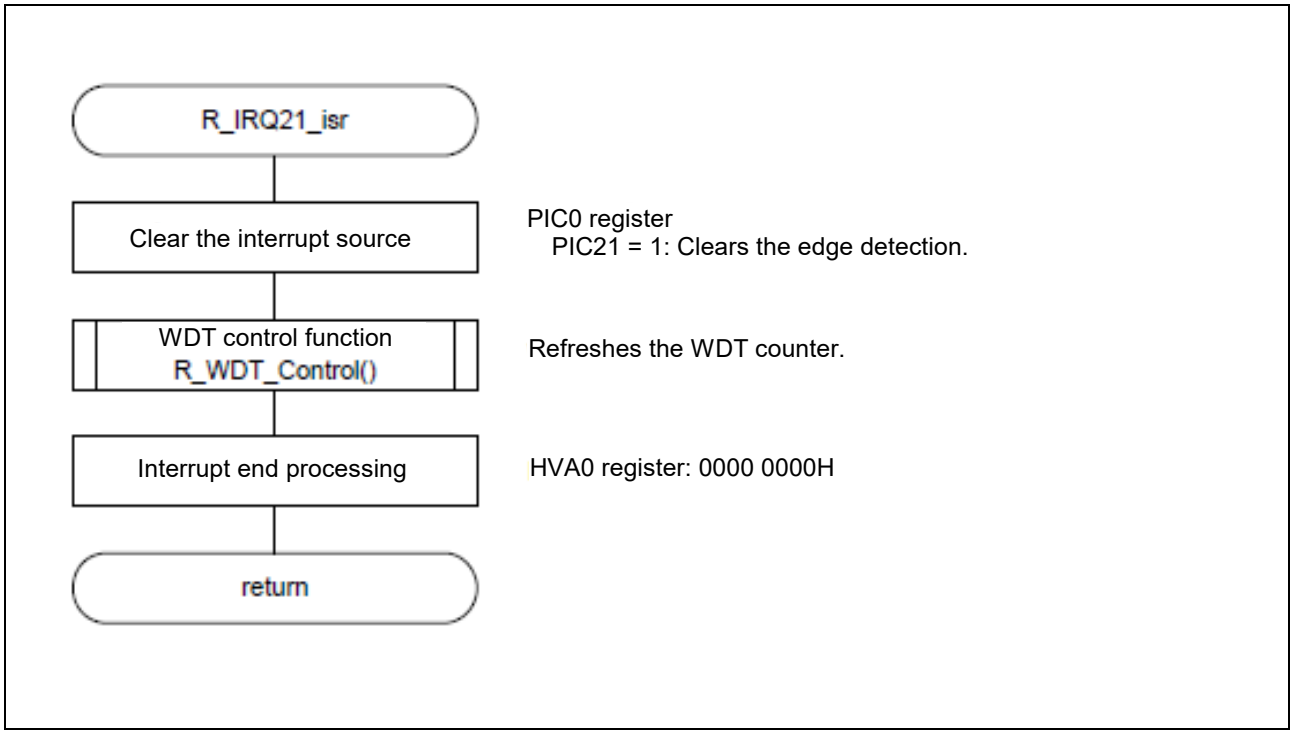


Figure 8-7 IRQ21 Interrupt (Compare Match Timer Channel 0 Interrupt) Processing

8.6 Tutorials

8.6.1 Operational Overview

This sample program makes initial settings for the watchdog timer (WDTA) and then refreshes the watchdog timer periodically (every 227.3 ms) based on interval interrupts from the compare match timer.

Switching the DIP switch (SW2 and SW3) generates external pin interrupt 2 in which the software wait processing is performed. Since CMT interval interrupts are suspended during this period, the watchdog timer (WDTA) refresh operation is stopped. For this reason, the counter value of the watchdog timer underflows, the error control module (ECM) is notified of an error, and the ECM is reset. LED2 is lit up in response to judgment that there has been a reset.

Table 8-3 Operational Overview summarizes the functions of this sample program. Figure 8-8 shows the timing diagram.

Table 8-3 Operational Overview

Function	Description
Communications channel	Channel 0 (WDT0) is used.
Clock	PCLKE/2048 (75 MHz/2048)
Timeout	16384 cycles (447.4 ms)
Window	Start: 75% End: 25%
ECM error notification	Enabled

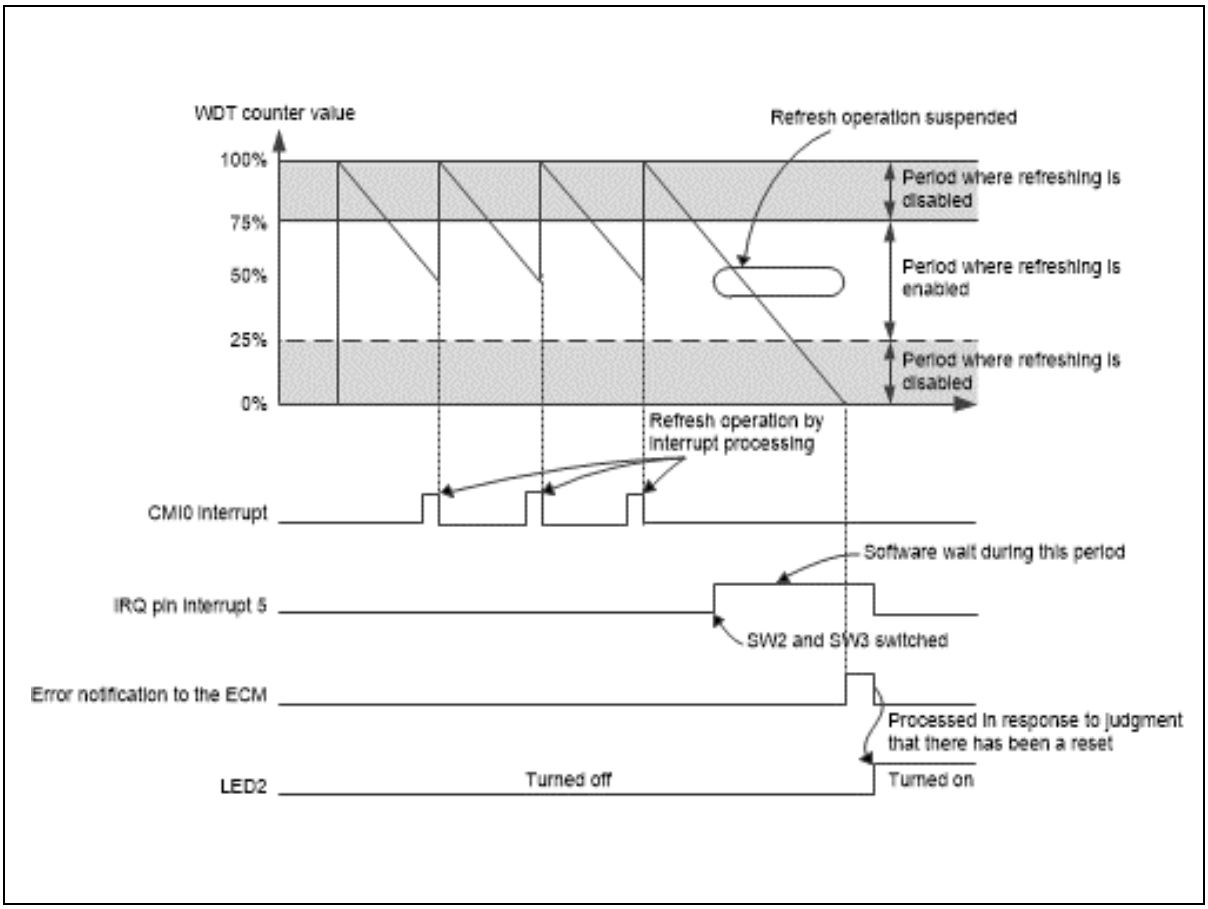


Figure 8-8 Timing Chart

9. IWDTA Sample Software

9.1 Overview

This section describes the sample program that controls resetting for the independent watchdog timer (IWDT) of the MCU mounted on the evaluation board.

The features of the IWDTA sample program are as follows.

- It refreshes the independent watchdog timer (IWDTA) periodically (every 273 ms) based on the compare match timer (CMT) after the independent watchdog timer starts operating.
- When a software wait is generated by an external interrupt (IRQ2), the refresh operation is stopped and a reset is generated. LED2 lights up after a reset.

Target Devices

EC-1

When applying the sample program covered in this application note to another microcontroller, modify the program according to the specifications for the target microcontroller and extensively evaluate the modified program.

9.2 Constants

Table 9-1 lists the constants used in this sample code.

Table 9-1 Constants Used in the Sample Program

Constant Name	Setting Value	Description
CMT0_CLOCK_PCLKD_512	3	Count clock = PCLKD/512
CMT0_CMI0_ENABLE	1	Enables CMI0 interrupts.
CMT0_INTERVAL_TIME	0x9C35	Sets the interval time to 273 ms.
CPG_CMT0_START	1	Starts CMT0 count.

9.3 Functions

Table 9-2 lists the functions used in this sample code.

Table 9-2 List of Functions

Function Name	Description	Scope	Definition File
main	Main processing	local	main.c
port_init	Initializes the port settings	local	main.c
ecm_init	Initializes the ECM settings	local	main.c
icu_init	Sets interrupts	local	main.c
cmt_init	Initializes the CMT module	local	main.c
iwdt_init	Initializes the IWDT	local	main.c
soft_wait	Software wait	local	main.c
R_IRQ2_isr	IRQ2 interrupt (IRQ pin interrupt 2) processing	local	main.c
R_IRQ21_isr	IRQ21 interrupt (compare match timer (CMI0)) processing	local	main.c

9.4 Details of the Functions

9.4.1 main

(1) **Synopsis**

Main processing

(2) **C language format**

void main (void);

(3) **Parameters**

None

(4) **Description**

This function is the main processing of the sample program:

- Initialization of the port settings.
- Initialization of the ECM.
- Initialization of the CMT.
- Initialization of the ICU.
- Initialization of the IWDT.

It makes the initial settings above to start CMT0. Then it repeats turning LED1 on and off in the main loop.

(5) **Returned values**

None

9.4.2 port_init

(1) Synopsis

Initializes the port settings.

(2) C language format

void port_init (void);

(3) Parameters

None

(4) Description

This function initializes the port settings.

(5) Returned values

None

9.4.3 ecm_init

(1) Synopsis

Initializes the ECM settings

(2) C language format

void ecm_init (void);

(3) Parameters

None

(4) Description

This function initializes the ECM settings.
It enables reset processing.

(5) Returned values

None

9.4.4 icu_init

(1) **Synopsis**

Sets interrupts.

(2) **C language format**

void icu_init (void);

(3) **Parameters**

None

(4) **Description**

This function enables interrupt processing.

(5) **Returned values**

None

9.4.5 cmt_init

(1) **Synopsis**

Initializes the CMT module.

(2) **C language format**

void cmt_init (void);

(3) **Parameters**

None

(4) **Description**

This function initializes channel 0 of the CMT.

(5) **Returned values**

None

9.4.6 iwdt_init

(1) **Synopsis**

Initializes the IWDT.

(2) **C language format**

void iwdt_init (void);

(3) **Parameters**

None

(4) **Description**

This function initializes the IWDT and starts counting by the IWDT.

(5) **Returned values**

None

9.4.7 soft_wait

(1) **Synopsis**

Software wait processing

(2) **C language format**

void soft_wait(void);

(3) **Parameters**

None

(4) **Description**

This function handles software wait processing by using the NOP command.

(5) **Returned values**

None

9.4.8 R_IRQ2_isr

(1) **Synopsis**

IRQ2 interrupt (IRQ pin interrupt 2) processing

(2) **C language format**

void R_IRQ2_isr (void);

(3) **Parameters**

None

(4) **Description**

This function handles software wait processing (about 3 seconds). Within this time, the watchdog timer will underflow and the ECM will be reset.

After release from the reset state, LED2 is lit up in response to judgment that there has been a reset.

(5) **Returned values**

None

9.4.9 R_IRQ21_isr

(1) **Synopsis**

IRQ21 interrupt (compare match timer (CMI0)) processing

(2) **C language format**

void R_IRQ21_isr (void);

(3) **Parameters**

None

(4) **Description**

This function refreshes the independent watchdog timer.

(5) **Returned values**

None

9.5 Flowcharts

9.5.1 Main Processing

The figure below is a flowchart of the main processing of the sample code.

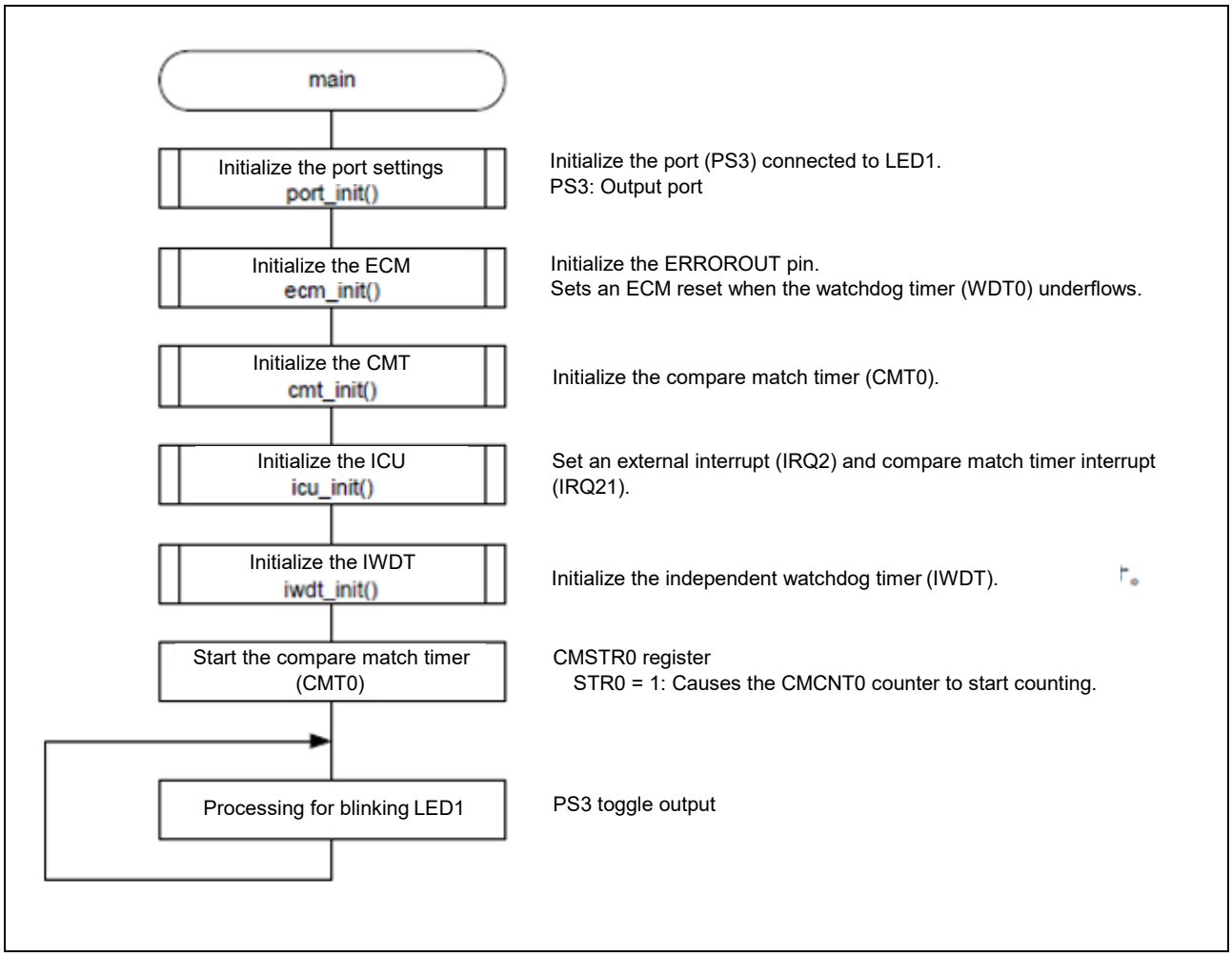


Figure 9-1 Main Processing of the Sample Code

9.5.2 Initialization of the IWDT

The figure below is a flowchart of processing for initialization of the IWDT.

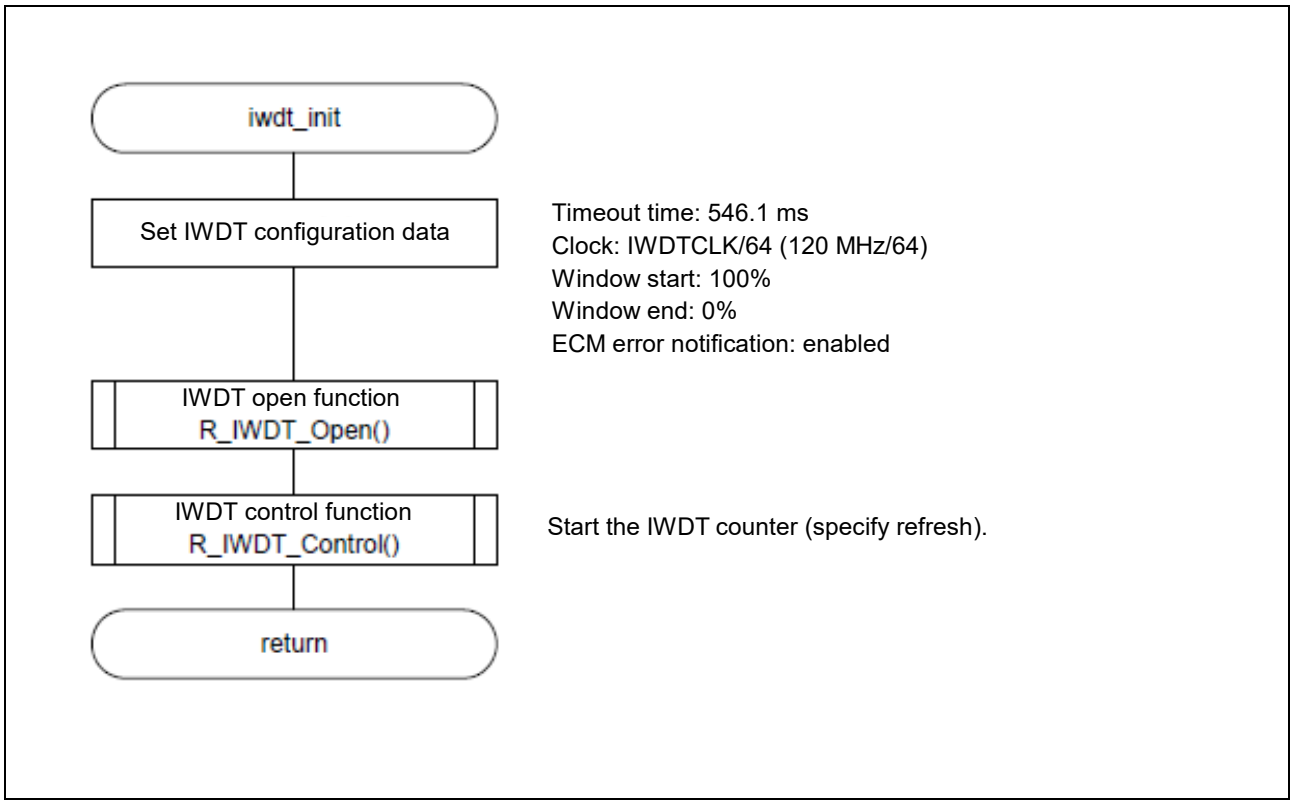


Figure 9-2 Initialization of the IWDT

9.5.3 IWDT Open Function

The figure below is a flowchart of the IWDT open function.

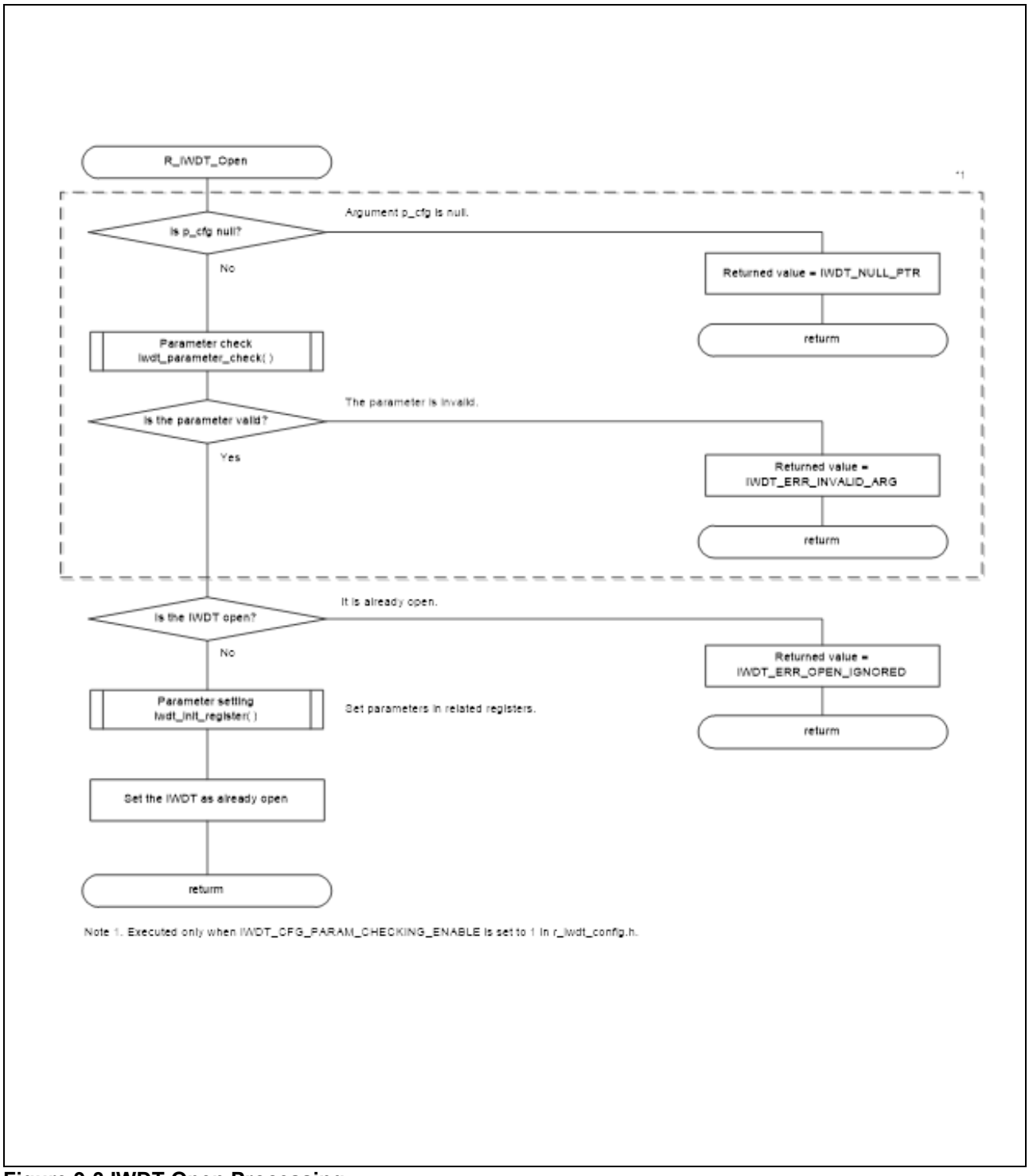


Figure 9-3 IWDT Open Processing

9.5.4 IWDT Control Function

The figure below is a flowchart of the IWDT control function.

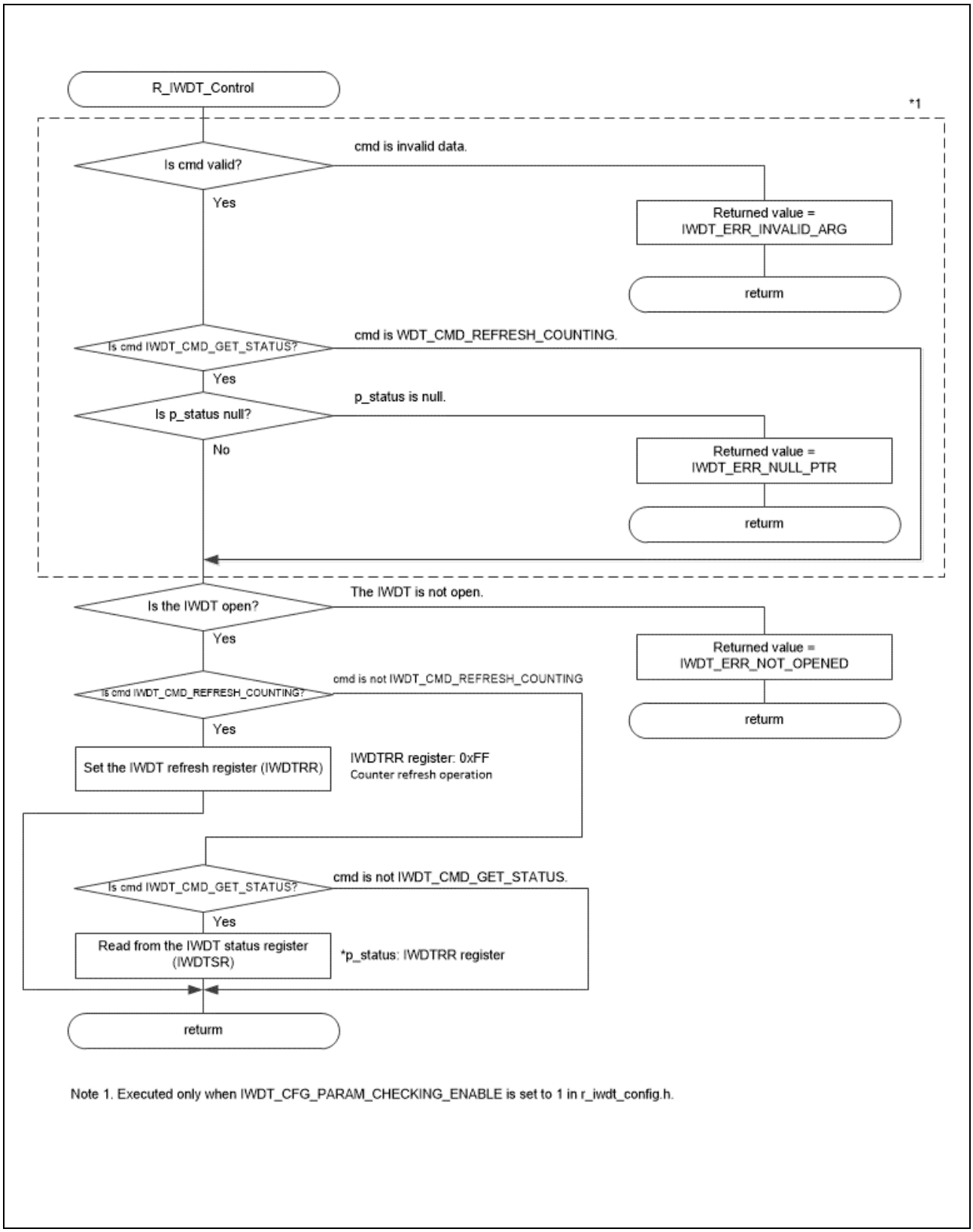


Figure 9-4 IWDT Control Processing

9.5.5 IRQ2 Interrupt (IRQ Pin Interrupt 2) Processing

The figure below is a flowchart of the IRQ2 interrupt (IRQ pin interrupt 2) processing.

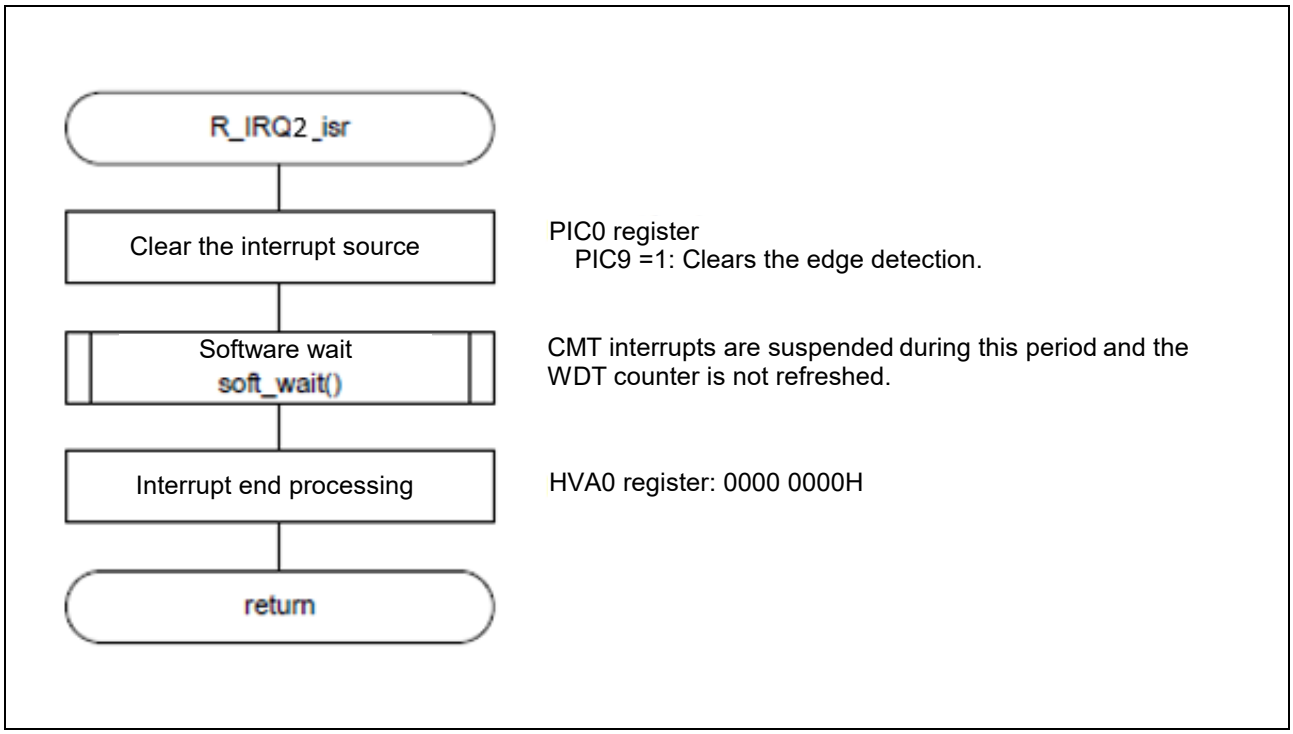


Figure 9-5 IRQ2 Interrupt (IRQ Pin Interrupt 2) Processing

9.5.6 IRQ21 Interrupt (Compare Match Timer 0 Interrupt) Processing

The figure below is a flowchart of the IRQ21 interrupt (compare match timer channel 0 interrupt) processing.

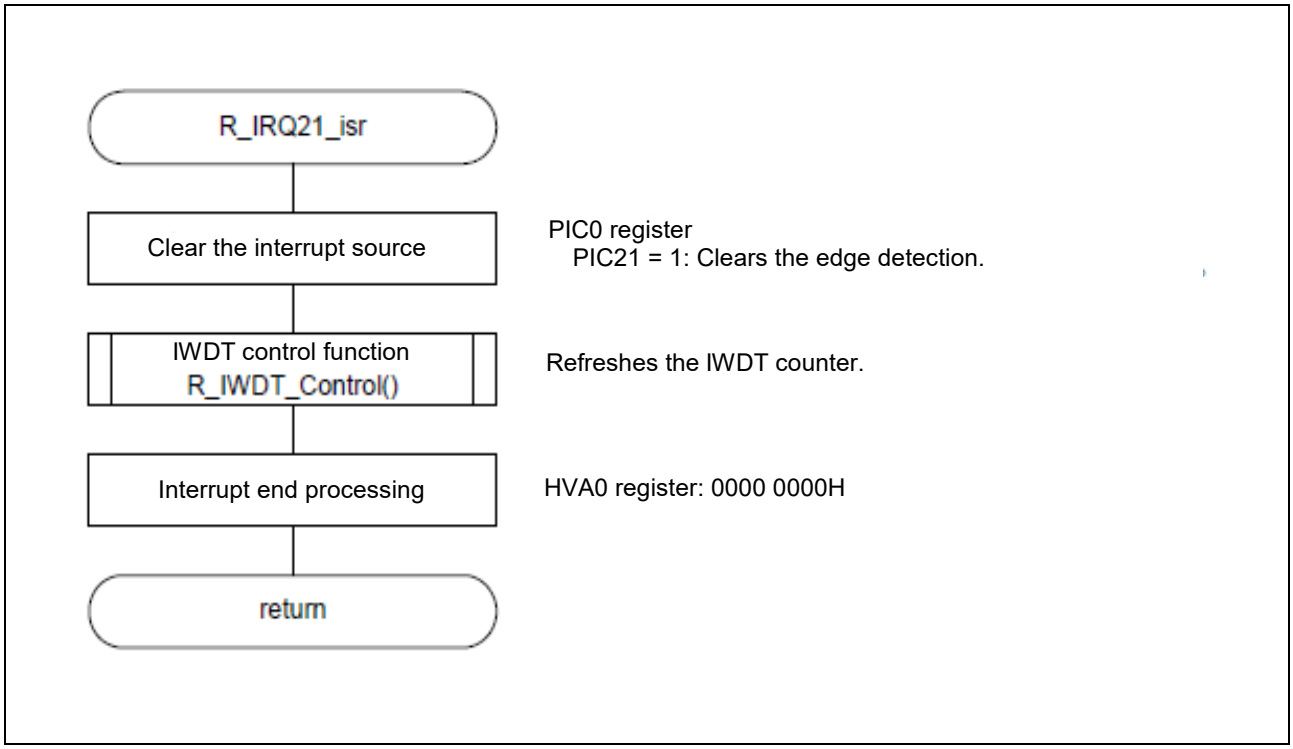


Figure 9-6 IRQ21 Interrupt (Compare Match Timer Channel 0 Interrupt) Processing

9.6 Tutorials

9.6.1 Operational Overview

This sample program makes initial settings for the independent watchdog timer (IWDTA) and then refreshes the watchdog timer periodically based on interval interrupts from the compare match timer.

Switching the DIP switch (SW2 and SW3) generates an external pin interrupt 2 in which the software wait processing is performed. Since CMT interval interrupts are suspended during this period, the independent watchdog timer (IWDTA) refresh operation is stopped. For this reason, the counter value of the independent watchdog timer will underflow, the error control module (ECM) will be notified of an error, and the ECM will be reset. After that, LED2 is lit up in response to judgment that there has been a reset.

Table 9-3 Operational Overview summarizes the functions of this sample program. Figure 9-7 shows the timing diagram.

Table 9-3 Operational Overview

Function	Description
Clock	IWDTCLK/64 (120 kHz/64)
Timeout	1024 cycles (546.1 ms)
Window	Start: 100% End: 0%
ECM error notification	Enabled

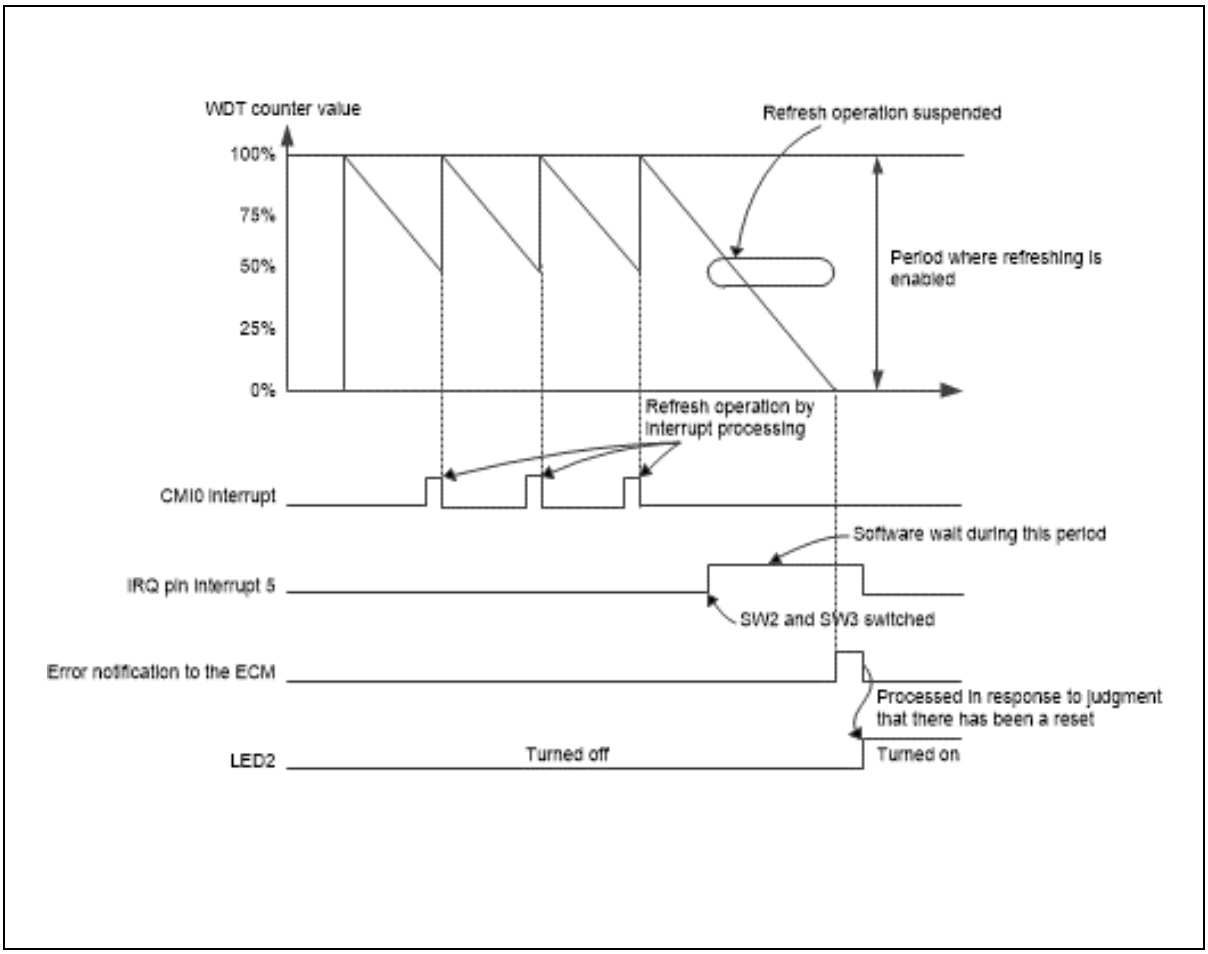


Figure 9-7 Timing Chart

10. RIIC Sample Software

10.1 Overview

This section describes the sample program that uses the I2C bus interface function (RIIC) of the MCU on the evaluation board to read from and write to the EEPROM (R1EX24016ASAS0G) mounted on the evaluation board.

The features of the RIIC sample program are as follows.

- It supports master transmission and master reception.
- It supports fast transfer mode (maximum transfer rate: 400 kbps)

Restrictions

The following restrictions apply to this sample program:

(1) It cannot be used in combination with the DMA.

(2) It does not support the timeout function of the RIIC.

(3) It does not support the NACK arbitration lost function of the RIIC.

(4) It does not support transmission of 10-bit addresses.

(5) It does not accept a restart condition for a slave device.

Do not specify the address of this module as the address immediately after a restart condition.

Target Devices

EC-1

When applying the sample program covered in this application note to another microcontroller, modify the program according to the specifications for the target microcontroller and extensively evaluate the modified program.

10.2 Constants

Table 10-1 lists the constants used in this sample code.

Table 10-1 Constants Used in the Sample Program

Constant Name	Setting Value	Description
WRITE_VAL	0xA5	Data values to be written to the EEPROM
PAGE_SIZE	0x10	The maximum number of bytes that can be written once
PORT_OUTPUT	0x3U	Setting value to the PDR register
PORT_PULL_UP_DOWN_NONE	0x0U	Setting value to the PCR register
PORT_GPIO_MODE	0x0U	Setting value to the PMR register
CMT_STOP	0x0U	Setting value to the CMSTR0 register (timer stop)
CMT_START	0x1U	Setting value to the CMSTR0 register (timer start)
CMT_INT_DISABLE	0x0U	Setting value to the CMCR0 register (interrupt disable)
CMT_INT_ENABLE	0x1U	Setting value to the CMCR0 register (interrupt enable)
CMT_CLOCK_SELECT	0x0U	Setting value to the CMCR0 register (division ratio of PCLKD)
CMT_COMPARE_VAL	9375U	Setting value to the CMCOR0 register (compare match period)
PRCR_WRITE_DISABLE	0x0000A500UL	Setting value to the PRCR register (MSTPCRA write protected)
PRCR_WRITE_ENABLE	0x0000A502UL	Setting value to the PRCR register (MSTPCRA write enabled)
MSTPCRA_CMT0_STOP	0x1U	Setting value to MSTPCRA register (CMT0 stop)
MSTPCRA_CMT0_START	0x0U	Setting value to MSTPCRA register (CMT0 start)
EEPROM_SIZE	2048U	EEPROM size
RIIC_CB_SUCCESS	0L	Notification driver successful completion
RIIC_CB_WAIT	1L	Wait driver callback
RIIC_CB_NACK	2L	Notification driver NACK
RIIC_CB_FAIL	-1L	Notification driver fail termination
CMT_CB_SUCCESS	0L	Notification driver successful completion
CMT_CB_WAIT	1L	Wait driver callback
ON	1U	Setting value to LED ON
OFF	0U	Setting value to LED OFF
LED1(x)	PORTS.PODR.BIT.B3 = (x)	Control LED1
LED2(x)	PORTS.PODR.BIT.B2 = (x)	Control LED2
LED3(x)	PORTS.PODR.BIT.B1 = (x)	Control LED3
LED4(x)	PORTS.PODR.BIT.B0 = (x)	Control LED4

Constant Name	Setting Value	Description
FAIL_SAFE_COUNT	0xFFFFFFFFU	Fail safe counter
HVA_DUMMY_DATA	0xC0FFEEU	Dummy data values to be written to inform the end-of-interrupt
RECEIVE_ACK	0L	-
RECEIVE_NACK	-1L	-

10.3 Functions

Table 10-2 lists the functions used in this sample code.

Table 10-2 List of Functions

Function Name	Description	Scope	Definition File
main	Main processing	local	main.c
ecm_init	Initializes the ECM settings	local	main.c
icu_init	Sets interrupts	local	main.c
isr_cmt	Initializes the CMT module	local	main.c
cb_riic	RIIC callback function	local	main.c
wait_riic_callback	Callback call wait processing	local	main.c
wait_1ms	Wait processing	local	main.c
ee_read	Reads data from EEPROM	local	main.c
ee_write	Writes data to EEPROM	local	main.c
init_led	Initializes the LED	local	main.c
wreite_read_check	Checks EEPROM data	local	main.c

10.4 Details of the Functions

10.4.1 main

(1) **Synopsis**

Main processing

(2) **C language format**

void main (void);

(3) **Parameters**

None

(4) **Description**

This function is the main processing of the sample program:
For details on the processing, see section 10.5.1, Main Processing.

(5) **Returned values**

None

10.4.2 ecm_init

(1) Synopsis

Initializes the ECM settings.

(2) C language format

void ecm_init (void);

(3) Parameters

None

(4) Description

This function initializes the ECM settings.

(5) Returned values

None

10.4.3 icu_init

(1) **Synopsis**

Sets interrupts.

(2) **C language format**

void icu_init (void);

(3) **Parameters**

None

(4) **Description**

This function enables interrupt processing.

(5) **Returned values**

None

10.4.4 isr_cmt

(1) **Synopsis**

Sets the CMT interrupt handler.

(2) **C language format**

static void isr_cmt(void)

(3) **Parameters**

None

(4) **Description**

This function sets the CMT interrupt handler.

(5) **Returned values**

None

10.4.5 cb_riic

(1) **Synopsis**

RIIC driver callback function

(2) **C language format**

static void cb_riic(void)

(3) **Parameters**

None

(4) **Description**

This function is the callback function called by the RIIC driver.

(5) **Returned values**

None

10.4.6 wait_riic_callback

(1) **Synopsis**

Callback call wait processing

(2) **C language format**

static int32_t wait_riic_callback(void)

(3) **Parameters**

None

(4) **Description**

This function waits until the callback function is called.

(5) **Returned values**

None

10.4.7 wait_1ms

(1) **Synopsis**

Wait processing

(2) **C language format**

static void wait_1ms(void)

(3) **Parameters**

None

(4) **Description**

This function waits until a callback is notified.

(5) **Returned values**

None

10.4.8 ee_read

(1) Synopsis

Reads data from the EEPROM.

(2) C language format

static riic_return_t ee_read(uint16_t address, uint32_t size, uint8_t *p_buf)

(3) Parameters

I/O	Parameter		Description
I	uint16_t	address	EEPROM address to start reading from
I	uint32_t	size	Read size (bytes)
I	uint8_t	*p_buf	Address of the buffer that stores the read data

(4) Description

This function is for reading data written to the EEPROM according to the argument value.
LED4 is lit up while data is being read.

(5) Returned values

Returned Value	Meaning
RIIC_SUCCESS	Processing ended successfully.
RIIC_ERR_LOCK_FUNC	The API is locked by another task.
RIIC_ERR_INVALID_CHAN	The channel does not exist.
RIIC_ERR_INVALID_ARG	The argument is invalid.
RIIC_ERR_OTHER	An event which is invalid because it is inapplicable in the current state has occurred.

10.4.9 ee_write

(1) Synopsis

Writes data to the EEPROM.

(2) C language format

static riic_return_t ee_write(uint16_t address, uint32_t size, uint8_t *p_buf)

(3) Parameters

I/O	Parameter		Description
I	uint16_t	address	EEPROM address to start writing to
I	uint32_t	size	Write size (bytes)
I	uint8_t	*p_buf	Address of the buffer to which data is written

(4) Description

This function is for writing data to the EEPROM according to the argument value.
LED3 is lit up while data is being written.

(5) Returned values

Returned Value	Meaning
RIIC_SUCCESS	Processing ended successfully.
RIIC_ERR_LOCK_FUNC	The API is locked by another task.
RIIC_ERR_INVALID_CHAN	The channel does not exist.
RIIC_ERR_INVALID_ARG	The argument is invalid.
RIIC_ERR_OTHER	An event which is invalid because it is inapplicable in the current state has occurred.

10.4.10 init_led

(1) **Synopsis**

Initializes the LED.

(2) **C language format**

static void init_led(void)

(3) **Parameters**

None

(4) **Description**

This function initializes the LED port settings.

(5) **Returned values**

None

10.4.11 write_read_check

(1) Synopsis

Checks EEPROM data.

(2) C language format

static int32_t write_read_check(void)

(3) Parameters

None

(4) Description

This function checks the read/write data and continues to light up LED2 if they match.
It turns the LED off if they do not match.

(5) Returned values

None

10.5 Flowcharts

10.5.1 Main Processing

The figure below is a flowchart of the main processing of the sample code.

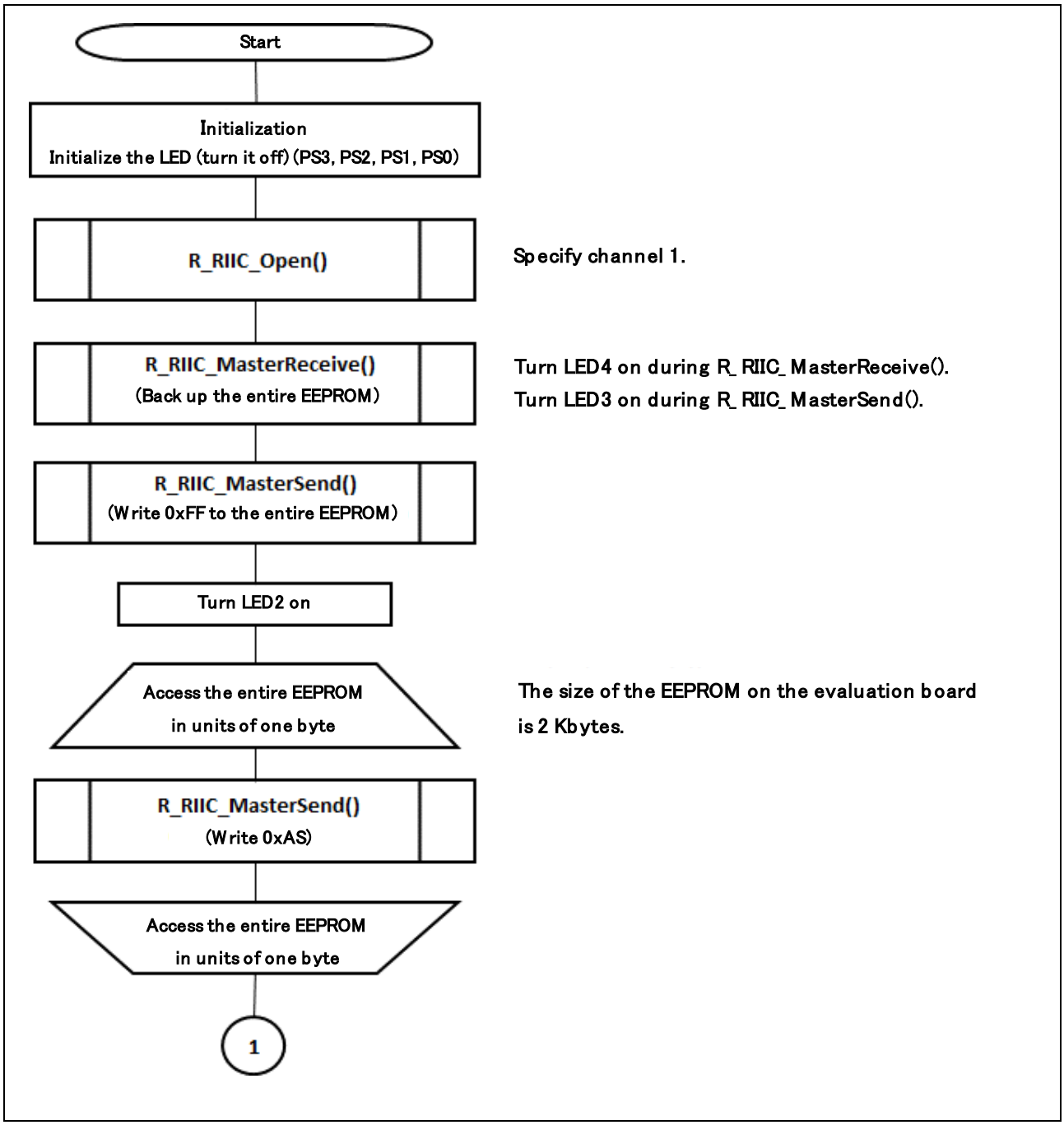


Figure 10-1 Main Processing of the Sample Code

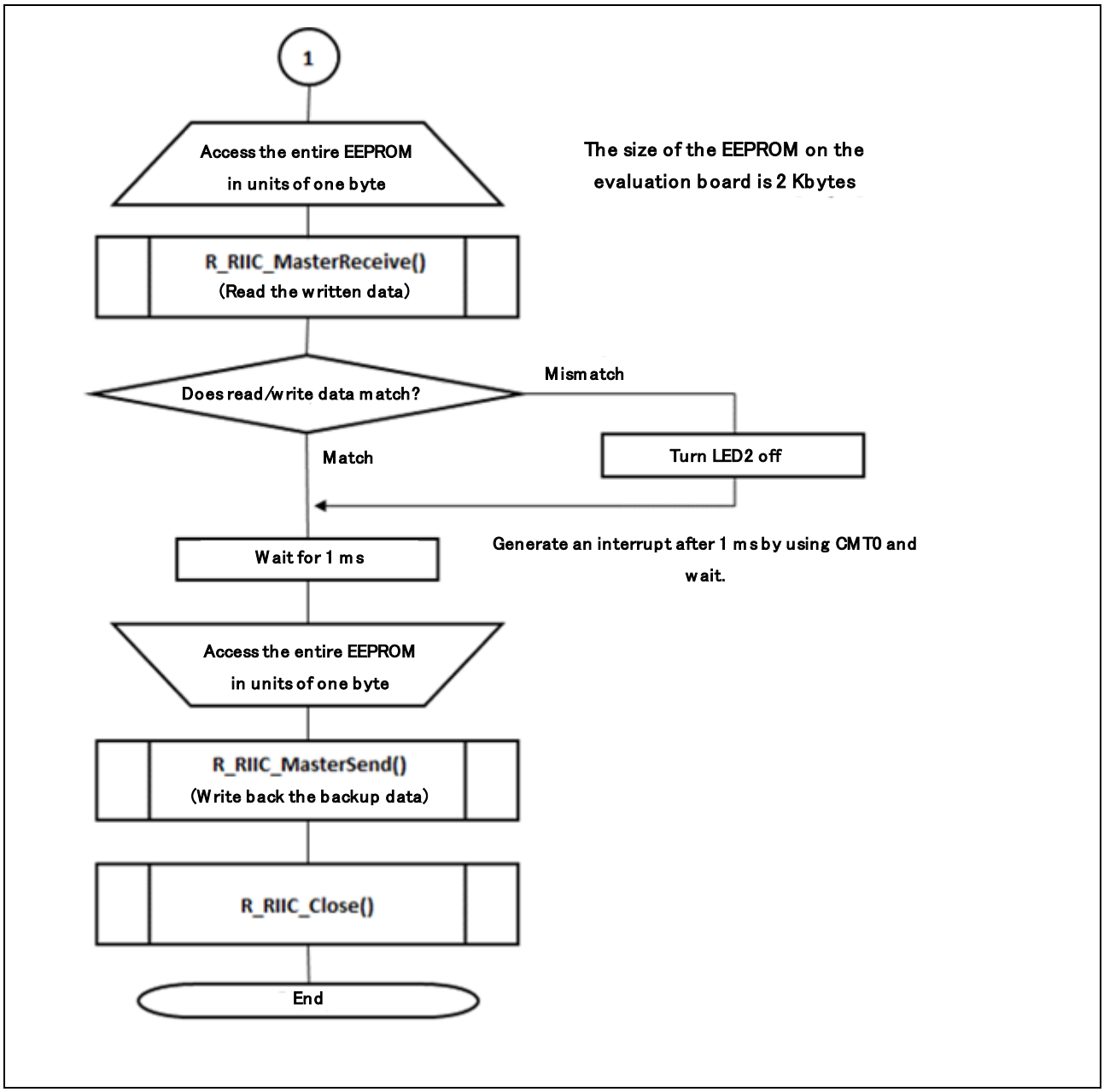


Figure 10-2 Main Processing of the Sample Code

10.5.2 Callback Processing

The figure below is a flowchart of the callback processing of the sample code.

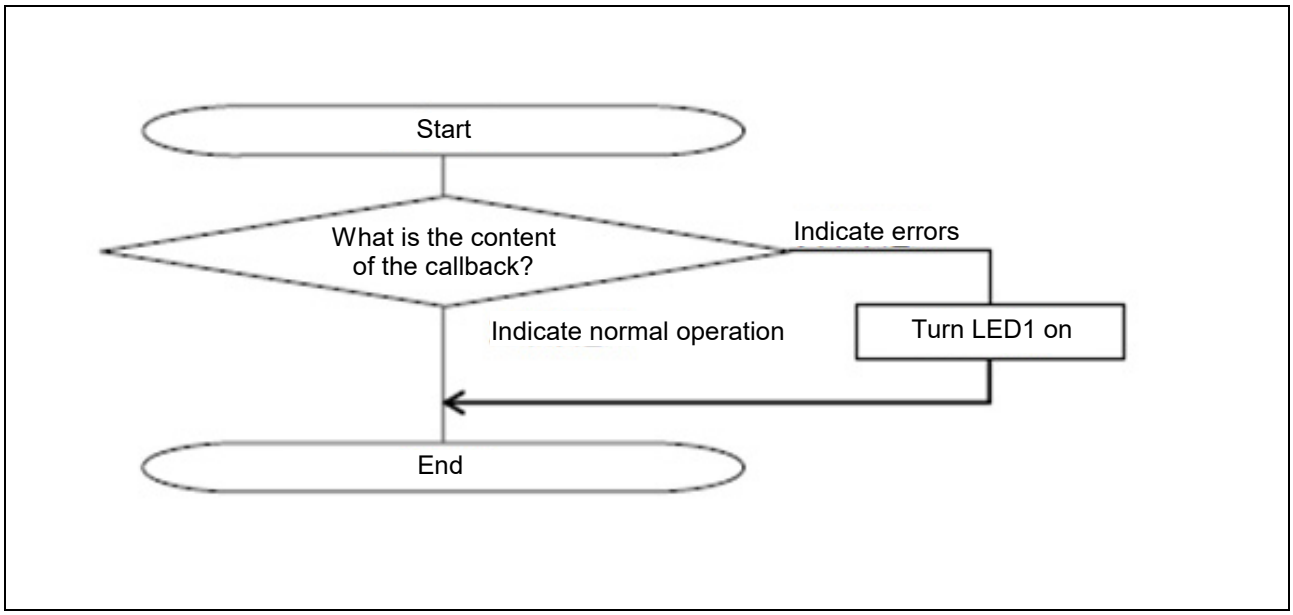


Figure 10-3 Callback Processing of the Sample Code

10.5.3 Compare Match Timer Interrupt Processing

The figure below is a flowchart of the compare match timer channel interrupt processing of the sample code.

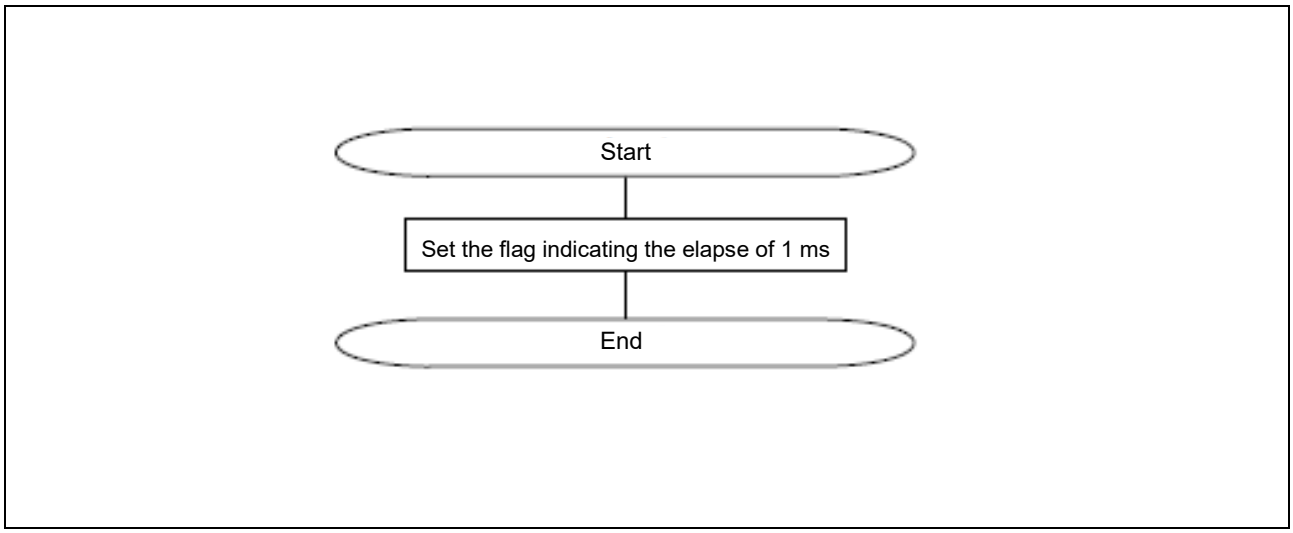


Figure 10-4 Compare Match Timer Interrupt Processing of the Sample Code

10.6 Tutorials

10.6.1 Operational Overview

Table 10-3 Operational Overview summarizes the functions of this RIIC sample program. Figure 10-5 shows a block diagram of the system.

Table 10-3 Operational Overview

Function	Description
Multiplexed pin settings	PC6 and PC7 are set for SCL1 and SDA1.
RIIC communications channel	Set channel 1 to which the EEPROM is connected.
Interrupt sources (interrupt priority)	• RIIC module transmission end (1)/reception end (1)/transmission buffer empty (1)/ error detection (1) • CMT module (for detecting 1 ms) compare match (15)
Transfer rate setting	400 kbps
Operating modes	Master transmission/reception
Operational overview	1. Back up the entire content of the EEPROM 2. Write 0xFF to the entire EEPROM 3. Write 0xA5 to the entire EEPROM 4. Check the written content 5. Write the content of the EEPROM back to the original state (Insert a 1-ms interval before a read/write access to the EEPROM)
Operation result display	• LED1 lights: A communications error in the RIIC was detected. • LED2 lights: The test pattern matched. • LED3 lights: The EEPROM is being written. • LED4 lights: The EEPROM is being read.

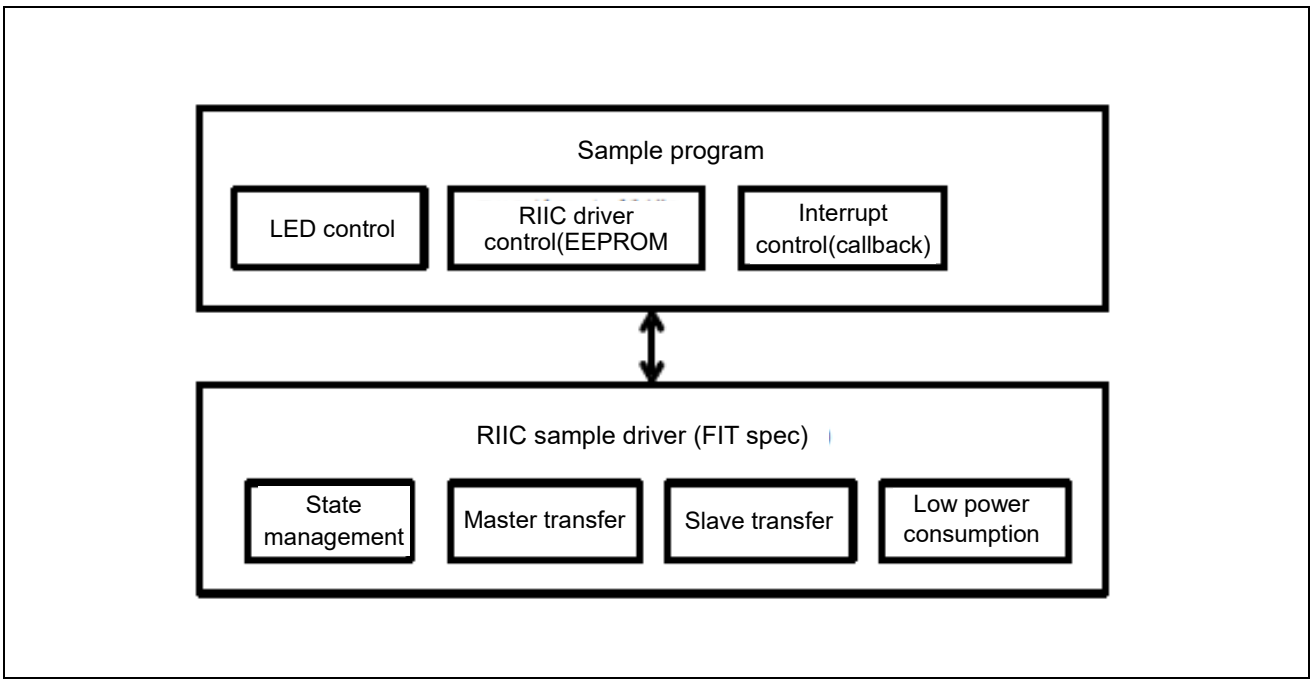


Figure 10-5 System Block Diagram

(1) **RIIC sample driver state transition diagram**

Figure 10-6 shows the state transitions of the RIIC sample driver.

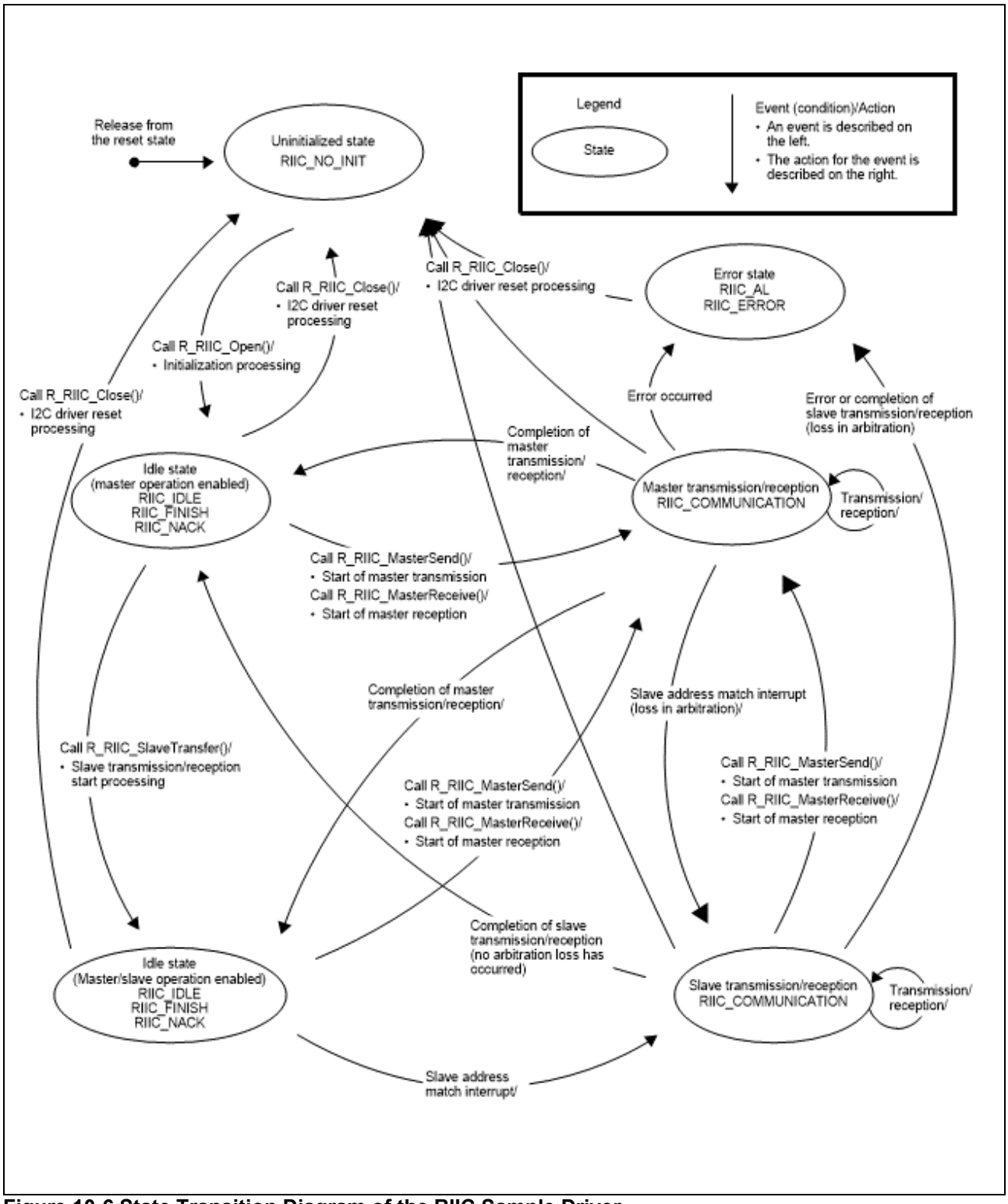


Figure 10-6 State Transition Diagram of the RIIC Sample Driver

(2) Flags at state transition of the RIIC sample driver

The I2C communications information structure includes device state flags (*dev_sts*) as its members. The state of communications of the device is stored in the device state flags.

These flags also allow multiple slave devices on the same channel to be controlled.

Table 10-4 lists device state flags when the state changes.

Table 10-4 Device State Flags at the time of State Transition

State	Device State Flag (dev_sts)
Uninitialized state	RIIC_NO_INIT
Idle state	RIIC_IDLE RIIC_FINISH RIIC_NACK
Communication underway (master transmission, master reception, slave transmission, slave reception)	RIIC_COMMUNICATION
Arbitration lost detection state	RIIC_AL
Error	RIIC_ERROR

(3) Arbitration lost detection function of the RIIC sample driver

This module can detect the following arbitration lost states. Although the RIIC can detect a loss in arbitration in slave transmission in addition to the following states, this module does not support it.

(i) If a start condition is issued while the bus is busy

This module detects a loss in arbitration if a start condition is issued when another master device already has issued a start condition and it occupies the bus (bus busy state).

(ii) If a start condition is issued later than another master device while the bus is not busy

This module tries to drive the SDA line low when issuing a start condition. However, if another master device issues a start condition earlier than this, the signal level on the SDA line does not match the level that this module outputs. At this time, this module detects a loss in arbitration.

(iii) If more than one start condition is issued at the same time

If multiple master devices issue a start condition at the same time, it may be recognized that a start condition has been completed normally on each master device.

Although each master device then starts communication, this module detects an arbitration lost state if the following conditions are met.

(a) If master devices transmit different data

This module compares the signal level on the SDA line with the level that this module outputs during data transfer. Therefore, if the level on the SDA line does not match that this module outputs during transmission of data including the slave addresses, this module detects a loss in arbitration at this point.

(b) If master devices transmit the same data in different number of times

If condition (a) above does not apply (the slave addresses and the data for transmission are the same), this module does not detect a loss in arbitration. It does detect a loss in arbitration if the number of data transmissions are different among master devices.

10.6.2 **Preparations**

The board must be modified to run the RIIC sample program.

(1) Connect the pins as follows so that the EEPROM on the board can be read and written.

- Between 2-pin J9 connector (SDA1 pin) and 5-pin U8 connector (SDA pin)
- Between 3-pin J9 connector (SCL1 pin) and 6-pin U8 connector (SCL pin)

(2) Connect pin 1 to pin 2 of the J10 connector to use the I2C.

Figure 10-7 shows the details of board modifications.

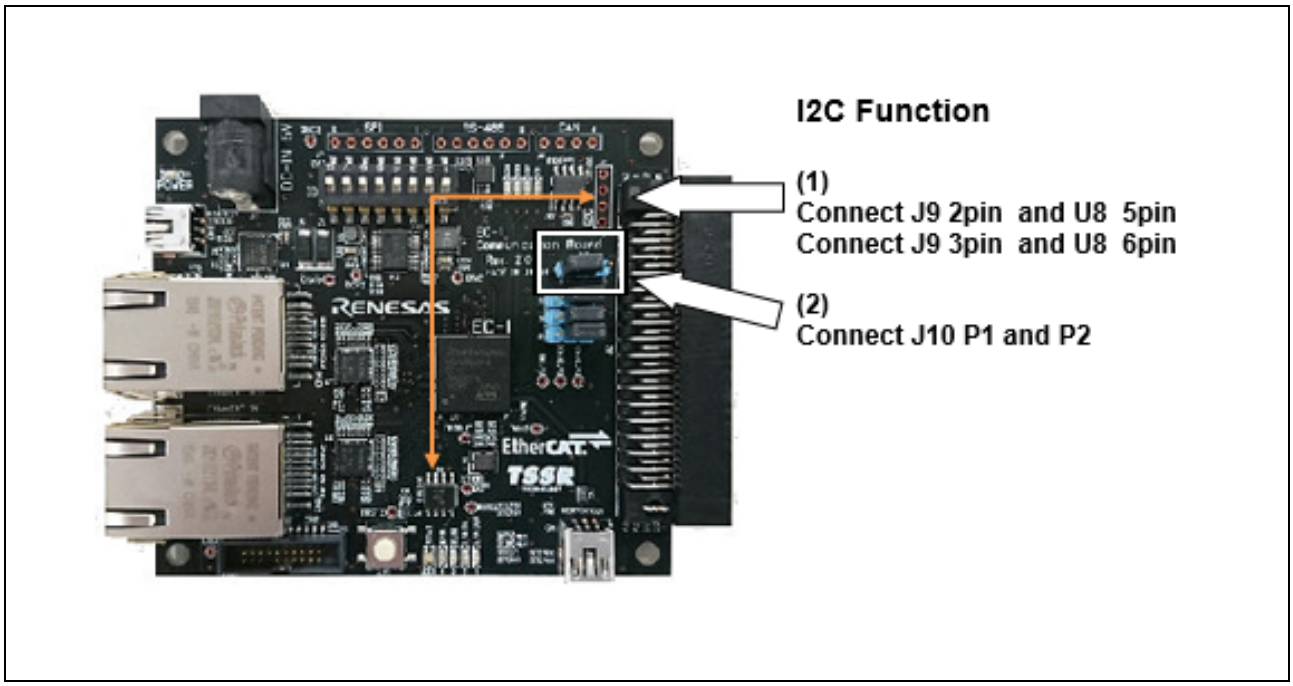


Figure 10-7 Board Modifications to Run the RIIC Sample Program

11. USB Host Sample Software

11.1 Overview

This section describes the sample program that uses the USB host driver that controls the USB host on the evaluation board.

The features of the USB host sample program are as follows.

The USB host sample software runs by combining the following modules (USB-BASIC firmware and HMSC).

(1) USB-BASIC firmware

The USB-BASIC firmware comprises the host driver (HCD), host manager (MGR), and the hub class driver (HUBCD) to control the hardware of the USB2.0HS host module.

The MGR manages the states of connected devices and performs enumeration. When the application (APL) changes the device state, call the corresponding API function from the APL to request a change of the state to the HCD or HUBCD.

The HUBCD is a sample program for enumerating and managing the states of devices connected to the downstream ports of a USB hub.

The USB-BASIC firmware runs in combination with the sample device class drivers provided by Renesas or the host device class drivers created by the user.

The firmware supports the following functions.

- Connection and disconnection of devices, suspension and resumption, and USB bus reset processing
- Control transfer, bulk transfer, interrupt transfer, and isochronous transfer
  - Full-speed/high-speed function device and enumeration (operating speed differs according to the device)
- Detecting transfer errors and retrying transfer
- Multiple device class drivers can be mounted without need to customize the firmware.

Restrictions

This firmware is subject to the following restrictions.

- Since the structures contain members of different types, addresses of the structure members may be misaligned by the complier.
- In host operation, the firmware does not support suspension and resumption of the connected hub and devices connected to the hub’s downstream ports.
- The firmware does not support suspension during data transfer. Confirm that the data transfer has been completed before executing suspension.
- Although a callback function is used to indicate overcurrent when it is detected, create the actual processing to suit your system.

For details of the USB-BASIC firmware, see section11.8, USB-BASIC Firmware..

(2) Host mass storage class driver (HMSC)

The HMSC runs as a USB host mass storage class driver in combination with the USB-BASIC firmware.

The HMSC is constructed by the BOT protocol of the USB mass storage class. By combining it with the file system, the HMSC can communicate with the USB storage device that supports the BOT. The driver is created based on the assumption that FatFs is used as the file system.

This module supports the following functions.

- Checking the connected USB storage device (whether it is ready to operate)
- Storage command communications by the BOT protocol
- Supports SFF-8070i (ATAPI) and SCSI as USB mass storage subclasses.
- Multiple USB storage devices can be connected.

Restrictions

This module is subject to the following restrictions.

- The structures contain members of different types (addresses of the structure members may be misaligned by the complier).
- The module only supports logical unit number 0 (LUN0).
- The module only supports the USB storage device whose sector size is 512 bytes.
- For a device that does not respond to the READ_CAPACITY command, the driver handles the sector size as 512 bytes.

For details of the HMSC, see section 11.9, Host Mass Storage Class Driver (HMSC).

Glossary

APL: Application program

ATAPI: AT Attachment Packet Interface

BOT: Mass storage class Bulk Only Transport

FSI: File System Interface

HCD: Host Control Driver of the USB-BASIC firmware

HMSC: Host Mass Storage Class driver

HMSCD: Host Mass Storage Class Driver unit

HMSDD: Host Mass Storage Device Driver

HUBCD: Hub Class Driver

LUN: Logical Unit Number

LBA: Logical Block Address

MGR: Peripheral device state manager of HCD

SCSI: Small Computer System Interface

Scheduler: Module for simple scheduling of task operations

Task: Unit of processing

USB-BASIC firmware: USB basic firmware for the EC-1 group

USB: Universal Serial Bus

Target Devices

EC-1

When applying the sample program covered in this application note to another microcontroller, modify the program according to the specifications for the target microcontroller and extensively evaluate the modified program.

11.2 Constants

The table below lists the constants used in this sample code.

Table 11-1 Constants Used in the Sample Program

Constant Name	Setting Value	Description
LED1	PORTS.PODR.BIT.B3	LED port setting
LED2	PORTS.PODR.BIT.B2	-
LED3	PORTS.PODR.BIT.B1	-
LED4	PORTS.PODR.BIT.B0	-
USER_DATA_SIZE	512	User-defined data size

11.3 Structures, Unions, and Enumerated Types

The table below lists the structures, unions, and enumerated types used in this sample code.

Table 11-2 DEV_INFO_t Structure

Member Name	Description
uint16_t state;	State for application
uint16_t devadr;	Device address

Table 11-3 STATE_t Enumerated Type

Member Name	Description
STATE_WAIT	Waiting for connection
STATE_DRIVE	Recognizing the drive
STATE_READY	Connecting to the drive
STATE_WRITE	Writing a file
STATE_READ	Reading a file
STATE_COMPLETE	Processing completed
STATE_ERROR	An error occurred

11.4 Functions

The table below lists the functions used in the USB host driver sample code.

Table 11-4 List of Functions

Function Name	Description	Scope	Definition File
main	Main processing	local	main.c
port_init	Initializes the port settings	local	main.c
icu_init	Sets interrupts	local	main.c
usbh_main	USB host main processing	local	r_usb_main.c
usb_cstd_PortInit	Initializes the port settings	local	r_usb_main.c
usb_cstd_IntInit	Sets interrupts	local	r_usb_main.c
usb_cstd_IntEnable	Enables interrupt processing	local	r_usb_main.c
usb_cstd_IntDisable	Disables interrupt processing	local	r_usb_main.c
PowerOnUSBh	Initializes the USB host	local	r_usb_main.c
AhbPciBridgeInit	Initializes the AHB-PCI bridge PCI communication register	local	r_usb_main.c
R_USBH_isr	USB host interrupt processing	local	r_usb_main.c
msc_main	MSC main processing	global	r_usb_hmsc_apl.c
msc_drive	Acquires information on the MSC drive	global	r_usb_hmsc_apl.c
msc_data_ready	Connects the MSC drive	global	r_usb_hmsc_apl.c
msc_data_write	Writes a file to the MSC drive	global	r_usb_hmsc_apl.c
msc_data_read	Reads a file from the MSC drive	global	r_usb_hmsc_apl.c
msc_configured	Processing upon the end of device enumeration	global	r_usb_hmsc_apl.c
msc_detach	Detach processing	global	r_usb_hmsc_apl.c
msc_suspend	Suspend processing	global	r_usb_hmsc_apl.c
msc_resume	Resume processing	global	r_usb_hmsc_apl.c
msc_drive_complete	Drive information acquisition end processing	global	r_usb_hmsc_apl.c
msc_init	Initializes the sample APL	global	r_usb_hmsc_apl.c
msc_registration	Registers the class driver	global	r_usb_hmsc_apl.c

11.5 Details of the Functions

11.5.1 main

(1) **Synopsis**

Main processing of the sample program

(2) **C language format**

void main (void);

(3) **Parameters**

None

(4) **Description**

This function is the main processing of the sample program.
It handles the following processing.

- Initialization of the ECM
- Setting interrupts
- Initialization of the port settings.
- Call the usbh_main() program

(5) **Returned values**

None

11.5.2 port_init

(1) **Synopsis**

Initializes the port settings.

(2) **C language format**

void port_init (void);

(3) **Parameters**

None

(4) **Description**

This function initializes the port settings.

(5) **Returned values**

None

11.5.3 icu_init

(1) **Synopsis**

Sets interrupts.

(2) **C language format**

void icu_init (void);

(3) **Parameters**

None

(4) **Description**

This function enables interrupt processing.

(5) **Returned values**

None

11.5.4 usbh_main

(1) **Synopsis**

USB host main processing

(2) **C language format**

void usbh_main (void);

(3) **Parameters**

None

(4) **Description**

This is the main processing of the USB host driver.

This function makes the initial settings for the USB driver.

- Calls the API function (R_usb_hstd_MgrOpen()) of the USB-BASIC firmware and registers the HCD and MGR tasks.
- Calls the API function (R_usb_hhub_Registration()) of the hub class driver and registers the hub task.
- After setting information in each member of the class driver registration structure (USB_HCDREG_t), calls the API function (R_usb_hstd_DriverRegistration()) of the USB-BASIC firmware and registers the class driver.
- Calls the API function (R_usb_hmsc_driver_start()) of the HMSC class driver and registers the HMSC and HSTRG tasks.

After the initial settings, this function acquires an event periodically in the main routine and processes the event.

(5) **Returned values**

None

11.5.5 usb_cstd_PortInit

(1) **Synopsis**

Initializes the port settings.

(2) **C language format**

void usb_cstd_PortInit (void);

(3) **Parameters**

None

(4) **Description**

This function initializes the port settings.

(5) **Returned values**

None

11.5.6 usb_cstd_IntInit

(1) **Synopsis**

Sets interrupts.

(2) **C language format**

void usb_cstd_IntInit (void);

(3) **Parameters**

None

(4) **Description**

This function handles USB host interrupt processing.

(5) **Returned values**

None

11.5.7 usb_cstd_IntEnable

(1) **Synopsis**

Enables USB host interrupt processing.

(2) **C language format**

void usb_cstd_IntEnable (void);

(3) **Parameters**

None

(4) **Description**

This function enables USB host interrupt processing.

(5) **Returned values**

None

11.5.8 usb_cstd_IntDisable

(1) **Synopsis**

Disables USB host interrupt processing.

(2) **C language format**

void usb_cstd_IntDisable (void);

(3) **Parameters**

None

(4) **Description**

This function disables USB host interrupt processing.

(5) **Returned values**

None

11.5.9 PowerOnUSBh

(1) Synopsis

Initializes the USB host.

(2) C language format

void PowerOnUSBh (void);

(3) Parameters

None

(4) Description

This function makes initial settings for the USB driver.

- I/O setting (set P66 as VBUSEN and P77 as OVERCUR)
- CPG setting (USB clock on, USB reset release)
- AHB-PCI bridge setting
- PHY internal PLL startup

(5) Returned values

None

11.5.10 AhbPciBridgeInit

(1) **Synopsis**

Initializes the AHB-PCI bridge PCI communication register.

(2) **C language format**

void AhbPciBridgeInit (void);

(3) **Parameters**

None

(4) **Description**

This function makes AHB-PCI bridge settings.

(5) **Returned values**

None

11.5.11 R_USBH_isr

(1) **Synopsis**

USB host interrupt processing

(2) **C language format**

void R_USBH_isr(void);

(3) **Parameters**

None

(4) **Description**

This function calls the interrupt handlers of the EHCI and OHCI while USB host interrupt processing is enabled.

(5) **Returned values**

None

11.5.12 msc_main

(1) Synopsis

MSC main processing

(2) C language format

void msc_main(void);

(3) Parameters

None

(4) Description

This function is the main processing of the MSC.
It handles processing according to the state by managing the MSC by state transition.
The flowchart of this function is shown in section 11.6.1, Application (APL).

(5) Returned values

None

11.5.13 msc_drive

(1) **Synopsis**

Acquires information on the MSC drive.

(2) **C language format**

uint16_t msc_drive(uint16_t drvno)

(3) **Parameters**

I/O	Parameter		Description
I	int16_t	drvno	Drive number

(4) **Description**

This function handles processing to make the required calls when the state is STATE_WRITE.

For the overview of the processing, see section 11.6.2, State Management.

(5) **Returned values**

Returned Value	Meaning
USB_TRUE	Success
USB_FALSE	Failure

11.5.14 msc_data_ready

(1) **Synopsis**

Connects the MSC drive.

(2) **C language format**

uint16_t msc_data_ready(uint16_t drvno)

(3) **Parameters**

I/O	Parameter		Description
I	int16_t	drvno	Drive number

(4) **Description**

This function handles processing to make the required calls when the state is STATE_READY.

For the overview of the processing, see section 11.6.2, State Management.

(5) **Returned values**

Returned Value	Meaning
USB_TRUE	Success
USB_FALSE	Failure

11.5.15 msc_data_write

(1) **Synopsis**

Writes a file to the MSC drive.

(2) **C language format**

uint16_t msc_data_write(uint16_t drvno)

(3) **Parameters**

I/O	Parameter		Description
I	int16_t	drvno	Drive number

(4) **Description**

This function handles processing to make the required calls when the state is STATE_WRITE.

For the overview of the processing, see section 11.6.2, State Management.

(5) **Returned values**

Returned Value	Meaning
USB_TRUE	Success
USB_FALSE	Failure

11.5.16 msc_data_read

(1) **Synopsis**

Reads a file from the MSC drive.

(2) **C language format**

uint16_t msc_data_read(uint16_t drvno)

(3) **Parameters**

I/O	Parameter		Description
I	int16_t	drvno	Drive number

(4) **Description**

This function handles processing to make the required calls when the state is STATE_READ.

For the overview of the processing, see section 11.6.2, State Management.

(5) **Returned values**

Returned Value	Meaning
USB_TRUE	Success
USB_FALSE	Failure

11.5.17 msc_configured

(1) **Synopsis**

Processing upon the end of device enumeration

(2) **C language format**

void msc_configured(uint16_t devadr)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	devadr	Driver address

(4) **Description**

This is a callback function that is called upon the end of enumeration after the MSC device is connected.

It calls drive open function R_usb_hmsc_StrgDriveOpen() and changes the state to STATE_DRIVE.

(5) **Returned values**

None

11.5.18 msc_detach

(1) **Synopsis**

Detach processing

(2) **C language format**

void msc_detach(uint16_t devadr)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	devadr	Device address

(4) **Description**

This is a callback function that is called when the USB device is disconnected.

It initializes the variables, releases the drive connected state and changes the state to STATE_WAIT.

(5) **Returned values**

None

11.5.19 msc_suspend

(1) **Synopsis**

Suspend processing

(2) **C language format**

void msc_suspend(uint16_t devaddr)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	devaaddr	Driver address

(4) **Description**

This is a callback function that is called when the USB device enters the suspended state.

It does not handle any special processing in the sample.

(5) **Returned values**

None

11.5.20 msc_resume

(1) **Synopsis**

Resume processing

(2) **C language format**

void msc_resume(uint16_t devaddr)

(3) **Parameters**

I/O	Parameter		Description
I	uint16_t	devaaddr	Driver address

(4) **Description**

This is a callback function that is called when the USB device is released from the suspended state.

It does not handle any special processing in the sample.

(5) **Returned values**

None

11.5.21 msc_drive_complete

(1) **Synopsis**

Drive information acquisition end processing

(2) **C language format**

void msc_drive_complete(USB_UTR_t *utr)

(3) **Parameters**

I/O	Parameter		Description
I	USB_UTR_t	*utr	Pointer to the structure

(4) **Description**

This is a callback function that is called upon the end of the drive information acquisition processing while the state is STATE_DRIVE.

It changes the state to STATE_READY.

(5) **Returned values**

None

11.5.22 msc_init

(1) **Synopsis**

Initializes the sample APL.

(2) **C language format**

void msc_init(void)

(3) **Parameters**

None

(4) **Description**

This function initializes the sample application.

It changes the state to STATE_WAIT.

(5) **Returned values**

None

11.5.23 msc_registration

(1) **Synopsis**

Registers the class driver.

(2) **C language format**

void msc_registration(void)

(3) **Parameters**

None

(4) **Description**

After setting information in each member of the class driver registration structure (USB_HCDREG_t), this function calls the API function (R_usb_hstd_DriverRegistration()) of the USB-BASIC firmware and registers the class driver.

(5) **Returned values**

None

11.6 Flowcharts

11.6.1 Application (APL)

The APL manages the USB host through the state transition.

Table 11-5 lists the states.

Table 11-5 List of States

State	Description
STATE_WAIT	Waiting for connection
STATE_COMPLETE	Detach processing
STATE_ERROR	An error occurred
STATE_DRIVE	Recognizing the drive
STATE_READY	Connecting to the drive
STATE_WRITE	Writing a file
STATE_READ	Reading a file

Figure 11-1 is a processing flowchart of the APL.

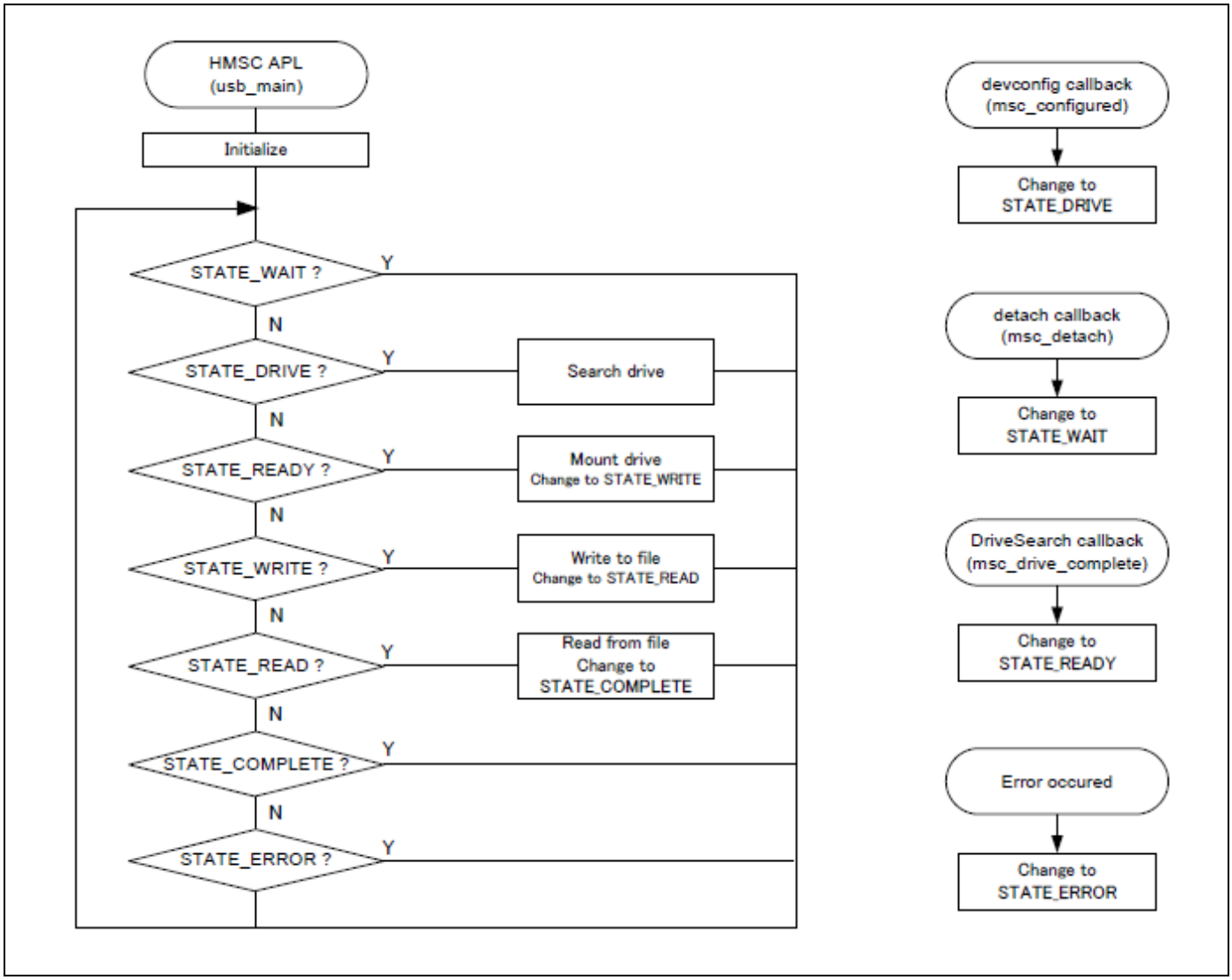


Figure 11-1 Main Processing of the Sample Code

11.6.2 State Management

This section outlines the processing of the APL in each state.

(1) Waiting for connection (STATE_WAIT)

In this state, the MSC device waits for connection. When enumeration is finished, the APL changes the state to STATE_DRIVE.

(1) The initialization function changes the state to STATE_READY.
(2) The APL remains in the STATE_WAIT state until the MSC device is connected.
(3) When the MSC device is connected and enumeration is finished, callback function msc_configured() set by member *devconfig* of the USB_HCDREG_t structure is called by the USB driver.
(4) Changes the state to STATE_DRIVE.

(2) Acquring information on the drive (STATE_DRIVE)

In this state, the APL acquires information on the connected MSC drive and changes the state to STATE_READY.

(1) Checks the flag variable (drive_search_lock) while recognizing the drive and starts processing if the variable is off.
(2) Sets drive_search_lock to ON.
(3) Calls R_usb_hmsc_StrgDriveSearch() and sends class request GetMaxLUN and the stage command to the MSC device to acquire information on the drive.
(4) When drive information acquisition processing is finished, calls callback function msc_drive_complete() registered in R_usb_hmsc_StrgDriveSearch().
(5) Changes the state to STATE_READY.

(3) Connecting the drive (STATE_READY)

In this state, the APL mounts the recognized drive and changes the state to STATE_WRITE.

(1) Calls f_mount() from the recognized driver number and connects the drive.
(2) Changes the state to STATE_WRITE.

(4) **Writing a file (STATE_WRITE)**

In this state, the APL writes a file to the connected drive and changes the state to STATE_READ.

(1) Calls f_open() and opens the file in the file creation + write mode.
(2) Calls f_write() and creates 512-byte file hmscdemX.txt which is all “a”. “X” in the file name corresponds to the drive number. For example, the file name for drive 1 is hmscdem1.txt.
(3) Calls f_close() and closes the file.
(4) Changes the state to STATE_READ.

(5) **Reading a file (STATE_READ)**

In this state, the APL reads a file from the connected drive and changes the state to STATE_COMPLETE.

(1) Calls f_open() and opens the file in the file creation + read mode.
(2) Calls f_read() and reads file hmscdemX.txt.
(3) Checks if the file contains 512 bytes which are all “a”.
(4) Calls f_close() and closes the file.
(5) Lights the LED corresponding to the drive number.
(6) Changes the state to STATE_COMPLETE.

(6) **End of processing (STATE_COMPLETE)**

The APL enters this state when the sample application processing is terminated normally.

(7) **Error termination (STATE_EEROR)**

The APL enters this state when the sample application processing is terminated abnormally.

(8) **Detach processing**

When the MSC device is disconnected, callback function msc_detach() is called by the USB driver. This callback function initializes the variables, releases the drive connected state, and changes the state to STATE_WAIT. Note that callback function msc_detach() is a function set in member *devdetach* in the USB_HCDREG_t structure.

11.7 Tutorials

11.7.1 Operational Overview

The main functions of sample application are as follows:

1. Performs enumeration processing when the MSC device is connected.
2. Acquires information on the drive when the enumeration processing of the MSC device is completed.
3. Writes a 512-byte file to the MSC device.
4. Reads the file written to the MSC device.
5. Lights the LED corresponding to the drive number if the compared file content matches.

Figure 11-2 shows the block diagram of the system.

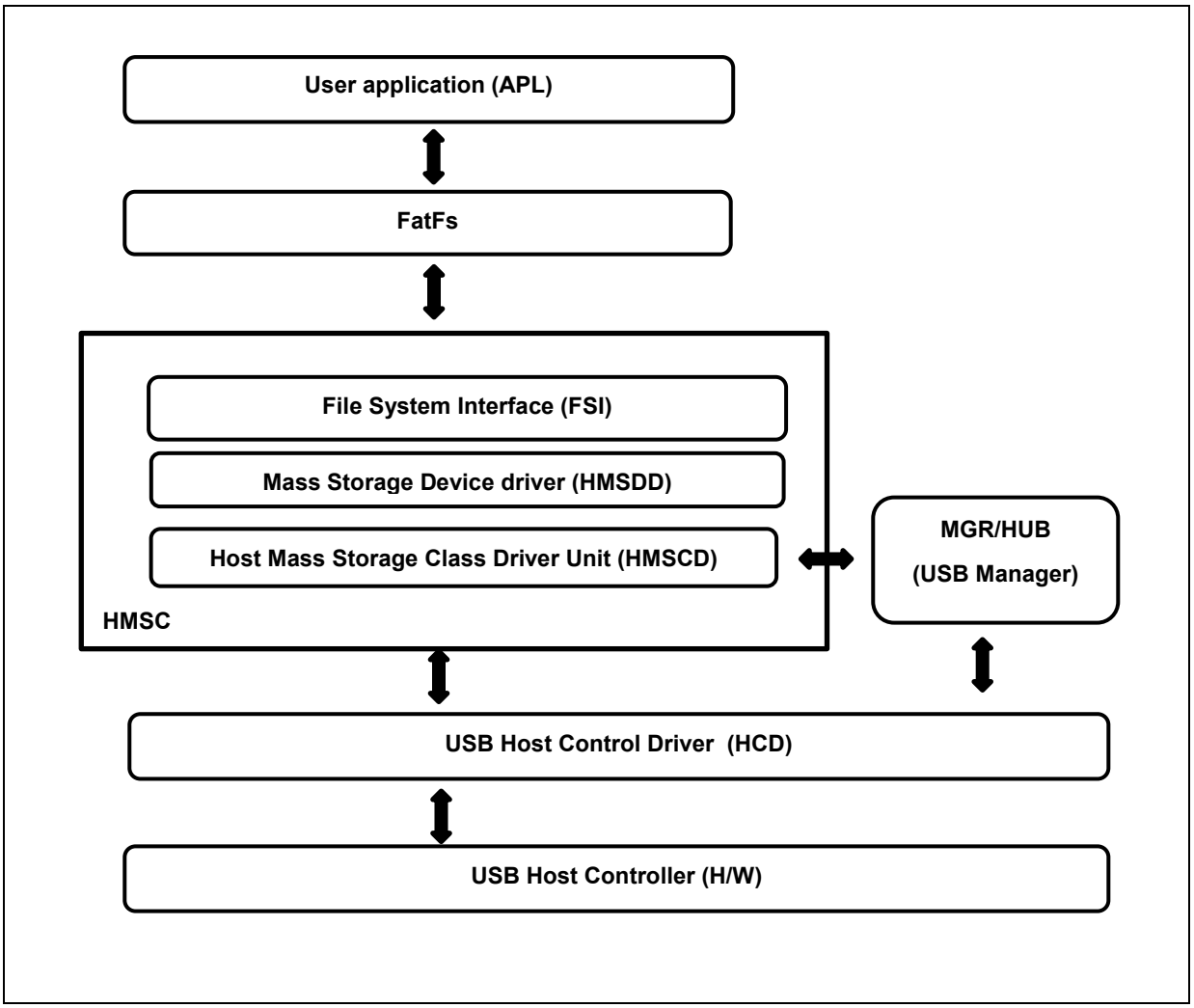


Figure 11-2 System Block Diagram

11.7.2 **Preparations**

Figure 11-3 is an example of the operating environment of the HMSC.

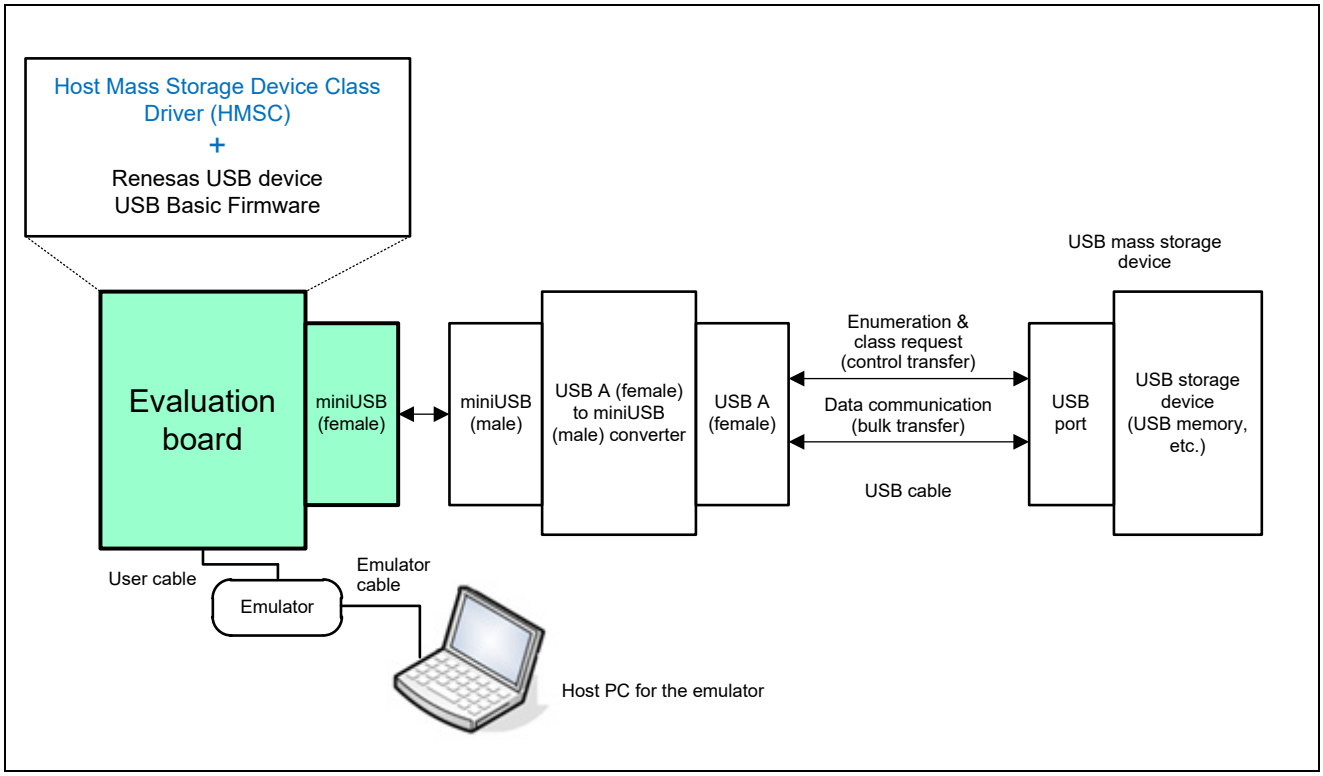


Figure 11-3 Connection Example

The board must be modified to run the USB host sample program.

(1) Although the USB host uses pin A16 as the overcurrent detection pin, this sample program does not use the overcurrent detection function, so connect 3.3 V and pin A16.

(2) Connect 5.0 V and the VBUS pin to supply power to the device as the USB host.

Figure 11-4 shows the details of board modifications.

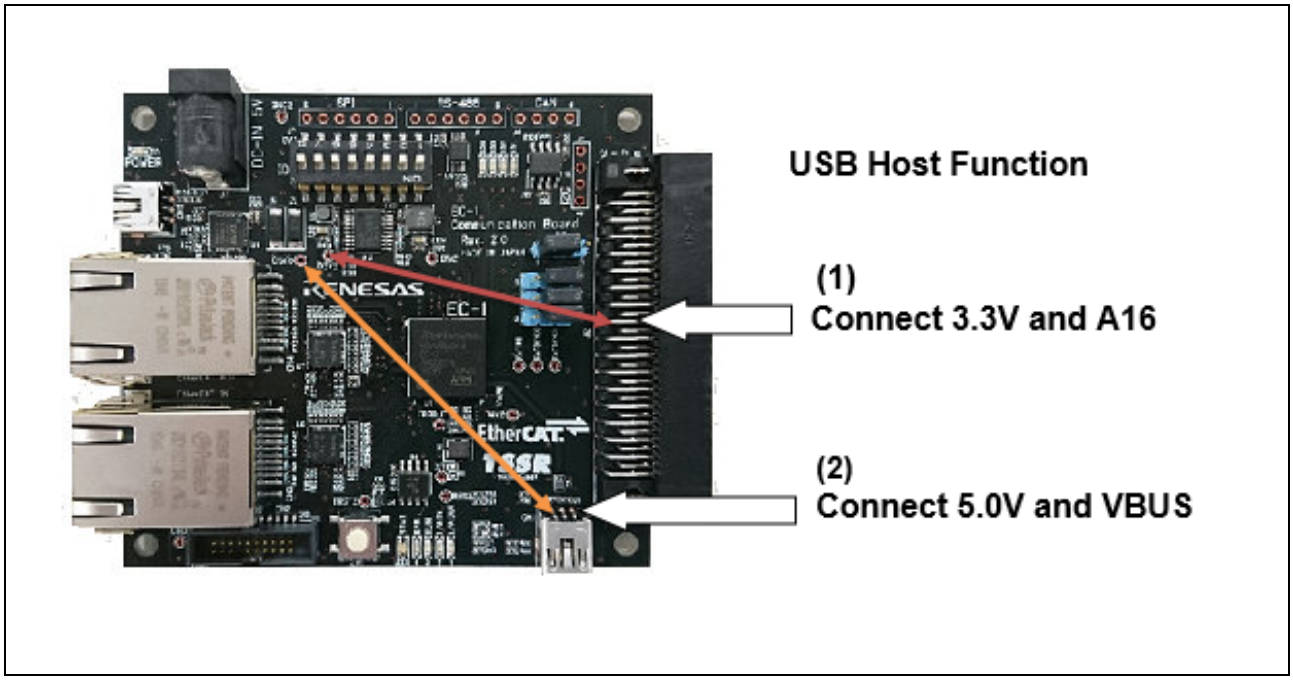


Figure 11-4 Board Modifications to Run the USB Host Sample Program

11.7.3 Sample Program Settings

Examples of initial settings are listed below.

```
void usb_hmsc_main(void)
{
  /* See the MCU settings */
  usb_mcu_setting();

  /* See USB driver settings */
  R_usb_hstd_MgrOpen();
  R_usb_cstd_SetTaskPri(USB_HUB_TSK, USB_PRI_3); // (See Note)
  R_usb_hhub_Registration(USB_NULL); // *
  msc_registration();
  R_usb_hmsc_driver_start();

  /* See the main routine */
  usb_hapl__mainloop();
}
```

Note: This function must be called only if the hub is used.

MCU settings

The USB module is set according to the initial setting sequence in the hardware manual, the USB interrupt handler is registered, and the USB interrupt enable setting is made.

USB driver settings

The USB driver settings register tasks to the scheduler and register class driver information for the USB-BASIC firmware. The following describes the procedure.

1. Call the API function (R_usb_hstd_MgrOpen()) of the USB-BASIC firmware and register the HCD and MGR tasks.
2. Call the API function (R_usb_hhub_Registration()) of the hub class driver and register the hub task.
3. After setting information in each member of the class driver registration structure (USB_HCDREG_t), call the API function (R_usb_hstd_DriverRegistration()) of the USB-BASIC firmware and register the class driver.
4. Call the API function (R_usb_hmsc_driver_start()) of the HMSC class driver and register the HMSC and HSTRG tasks.

The following is an example of information to be set in the structure declared in USB_HCDREG_t.

```
void msc_registration(void)
{
  /* Structure for registering the class driver */
  USB_HCDREG_t driver;
  /* Set a class code defined in the USB standard */
  driver.ifclass = (uint16_t)USB_IFCLS_MSC;
  /* Set a target peripheral list */
  driver.tpl = (uint16_t*)&usb_gapl_devicetpl; *1
  /* Set a class check function executed during enumeration */
  driver.classcheck = &R_usb_hmsc_class_check;
  /* Set the function called when enumeration is finished */
  driver.devconfig = &msc_configured;
```

```
    /* Set the function called when the USB device is disconnected */
    driver.devdetach = &msc_detach;
    /* Set the function called when the device enters the suspended state */
    driver.devsuspend = &msc_suspend;
    /* Set the function called when the device is released from the suspended state */
    driver.devresume = &msc_resume;
    /* Register class driver information to the HCD */
    R_usb_hstd_DriverRegistration(&driver);
}
```

Note 1. Define the TPL (Target Peripheral List) in the application.
For details, see section 11.8.6, Target Peripheral List (TPL).

Main routine

After the initial settings, the USB driver operates by calling the scheduler (R_usb_cstd_Scheduler()) in the main routine of the application.

Whether or not an event is generated is checked by calling R_usb_cstd_Scheduler() in the main routine. If an event is generated, a flag is set to notify the scheduler that an event is generated.

After calling R_usb_cstd_Scheduler(), call R_usb_cstd_CheckSchedule() to check if an event is generated. Note that event acquisition and the event processing must be performed periodically.*1

```
void usb_hapl_mainloop(void)
{
    while(1) /* Main routine */
    {
        /* Check and acquiring an event and set a flag *1 */
        R_usb_cstd_Scheduler();
        /* Check if an event is generated and clear the flag */
        if(USB_FLGSET == R_usb_cstd_CheckSchedule())
        {
                R_usb_hstd_MgrTask(); /* MGR task */
                R_usb_hhub_Task(); /* HUB task *3 */
                R_usb_hmsc_task(); /* HMSC task */
                R_usb_StrgDriveTask(); /* HSTRG task */
        }
        msc_main(); /* APL */
    }
}
```

*2

Note 1. If, after acquiring an event by R_usb_cstd_Scheduler(), another event is acquired by R_usb_cstd_Scheduler() again before processing, the first event will be discarded. After acquiring the event, call each task and perform processing.
Note 2. Describe these processing in the main loop of the application program.
Note 3. This function must be called only if the hub is used.

11.7.4 Incorporating FatFs

To build this sample program, the user must incorporate FatFs.

This section describes how to incorporate FatFs.

FatFs is released to the public as an open source in the following URL:

http://elm-chan.org/fsw/ff/00index_e.html

1. If you scroll the page, you will find the link for downloading in the Resources section.
2. Click FatFs R0.13, download FatFs and store it in a folder.

This sample program is created for version R0.13. If FatFs is already updated, find this version from the Old release links.

Unzip the downloaded FatFs file (ff13.zip) and move it to the work space of the sample program as shown in Figure 11-5.

Note that this sample program runs only in this folder structure.

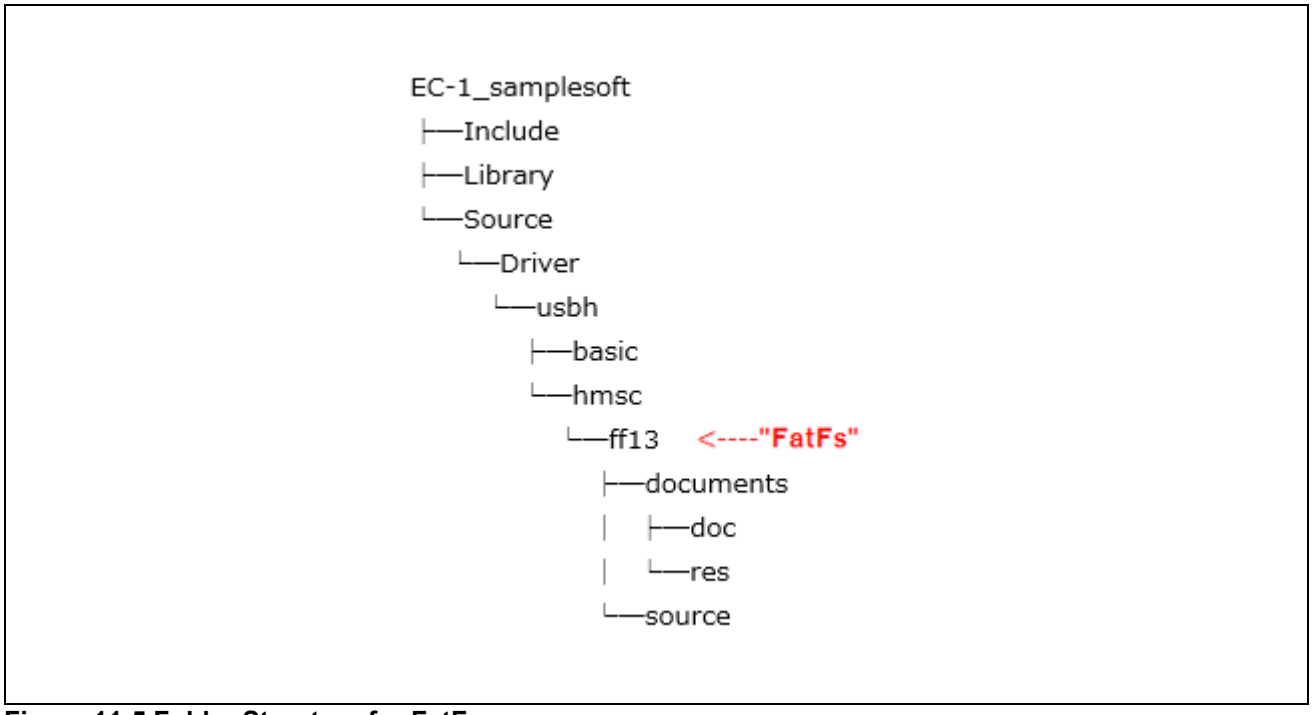


Figure 11-5 Folder Structure for FatFs

Open the work space of the sample program and build the sample project.

The sample program is set to link to FatFs, so the setting does not have to be changed.

To incorporate FatFs into the user's product, check the license of FatFs and incorporate FatFs on the user's own responsibility.

11.8 USB-BASIC Firmware

11.8.1 Scheduler Function

This firmware uses a scheduler to manage requests generated by tasks and hardware according to their relative priority. When multiple task requests with the same priority are generated, a FIFO queue determines their order of execution.

Requests between the tasks are implemented by transmitting and receiving messages. In addition, callback functions are used for responses to tasks indicating the end of a request, so the user need only install appropriate class drivers for the system and there is no need to modify the scheduler itself.

For details of the schedule function, see section 11.8.10, Non-OS Scheduler.

11.8.2 Host Control Driver (HCD)

- Basic functions

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

1. Control transfer (ControlRead/ControlWrite/No-dataControl) and notification of results
2. Data transfer (Bulk/Interrupt/Isochronous) and notification of results
3. Forced end of data transfer (all pipes)
4. Error detection in USB communications and retrying transfer
5. Sending request signals for the USB bus and notification of the results of the reset handshake
6. Transmission or suspend and resume signals
7. Generation of interrupts to indicate the detection of attachment and detachment

- Issuing a request to the HCD

For requests to update the state of the connected devices, control may be exercised by the HCD either directly or via a USB hub. The MGR determines the device address and issues control requests to the HCD and HUBCD tasks.

In response to requests from the higher layer task, the HCD sends notifications of the results by means of callback functions.

- When two or more devices are connected

Enumeration proceeds with the device address of the connected hub as 1. In addition, the HUBCD automatically assigns device addresses of 2 and higher to the devices connected to the downstream ports.

Enumeration and communications with two or more devices are possible, but the HUBCD cannot enumerate multiple devices simultaneously. In such cases, after the first connected device has been enumerated, enumeration of the next connected device starts, and so on.

Furthermore, the HUBCD does not calculate the USB bus occupancy ratios.

11.8.3 **Host Device Class Control Driver (HDCD)**

- Registration

A HDCD suited to the user system must be created.

Register various information on the created HDCD in the USB_HCDREG_t structure by using the API function R_usb_hstd_DriverRegistration().

For details of the method of registration, see section 3.14.3.

- Setting tasks

When the HDCD uses a scheduler, see section 11.8.10, Non-OS Scheduler.

Add a task ID, mailbox ID and memory pool ID for each added task in the r_usb_basic_config.h file.

Set the task priority by using the API function R_usb_cstd_SetTaskPri().

Note the following when setting these items:

- Assign consecutive ID numbers to tasks, and do not assign the same ID to more than one task.
- Set the same value as task ID for mailbox ID and memory pool ID.
- Set the priority of added tasks to a value lower than that of HCD(1), MGR(2), or HUB(3) (4 to 7), with 0 being the highest and 7 being the lowest.

Examples of additional settings in r_usb_basic_config.h are listed below.

#define USB_SMP_TSK USB_TID_6: Task ID

#define USB_SMP_MBX USB_SMP_TSK: Mailbox ID

#define USB_SMP_MPL USB_SMP_TSK: Memory pool ID

If necessary, set the following items in the r_usb_basic_config.h file.

For details, see section 3.2.11, USB.

- Maximum value of task ID
- Maximum number of memory pools
- Maximum number of message pools

- Class check processing

Use the class check callback function for class checking during enumeration.

On completion of processing, call the API function R_usb_hstd_ReturnEnuMGR() to convey the result.

11.8.4 **Host Manager (MGR)**

- Basic functions

The MGR is a task that provides supplementary functionality between the HCD and HDCD. The functions of the MGR are listed below.

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

- USB standard requests

The MGR enumerates connected devices.

The USB standard requests issued by the MGR are listed below. The descriptor information fetched from a device is stored internally and can be registered as pipe information by using the API function.

GET_DESCRIPTOR (Device Descriptor)

SET_ADDRESS

GET_DESCRIPTOR (Configuration Descriptor)

SET_CONFIGURATION

- Class checking

The MGR notifies the HDCD of the information fetched by the GET_DESCRIPTOR request during enumeration, and checks if the connected device is ready to operate.

Enumeration continues according to the response returned by the API function R_usb_hstd_ReturnEnuMGR() from the HDCD.

11.8.5 **Hub Class Driver (HUBCD)**

- Basic functions

The HUBCD manages the states of the downstream ports of a connected USB hub and provides supplementary functionality between the HCD and HDCD

The functions of the HUBCD are listed below.

1. Enumeration of the USB hub
2. State management for devices connected to the downstream ports of a USB hub
3. Enumeration of devices connected to the downstream ports of a USB hub

- State management of the downstream ports

When the USB hub is connected, the HUBCD performs the following for all the downstream ports to check the device connection state of each downstream port.

(1) Enables the port power (HubPortSetFeature: USB_HUB_PORT_POWER).

(2) Initializes the ports (HubPortClrFeature: USB_HUB_C_PORT_CONNECTION).

(3) Acquires the port status (HubPortStatus: USB_HUB_PORT_CONNECTION).

- Connecting devices to the downstream ports

When the HUBCD receives notification of a device being attached to a downstream port of the USB hub, it issues a USB reset signal request to the USB hub (HubPortSetFeature: USB_HUB_PORT_RESET).

After that, it uses the MGR task resource to enumerate the connected devices.

The HUBCD stores the results of the reset handshake and assigns successive unused device addresses to the connected devices.

- Class request

Table 11-6 lists the class requests that the HUBCD supports.

Table 11-6 USB Hub Class Requests

Request	Implementation	Function Name	Description
ClearHubFeature	No		
ClearPortFeature	Yes	usb_hhub_PortClrFeature	USB_HUB_C_PORT_CONNECTION USB_HUB_C_PORT_RESET
ClearTTBuffer	No		
GetHubDescriptor	Yes	usb_hhub_GetHubDescriptor	Acquires the device descriptor
GetHubStatus	No		
GetPortStatus	Yes	usb_hhub_GetStatus	Acquires the port status
ResetTT	No		
SetHubDescriptor	No		
SetHubFeature	No		
SetPortFeature	Yes	usb_hhub_PortSetFeature	USB_HUB_PORT_POWER USB_HUB_PORT_RESET
GetTTState	No		
StopTT	No		

11.8.6 **Target Peripheral List (TPL)**

Which USB devices the USB-BASIC firmware supports is specifiable.

Specify the supported USB devices in sets as the product IDs and vendor IDs of the USB devices in the target peripheral list (TPL).

If you need not specify supported USB devices, set USB_NOVENDOR and USB_NOPRODUCT for the vendor ID and product ID, respectively.

An example of how to create a TPL is shown below.

```
const uint16_t usb_gapl_devicetpl[] =
{
    2, /* Number of list */
    0, /* Reserved */
    0xXXXX, /* Vendor ID */
    0xXXXX, /* Product ID */
    0xYYYY, /* Vendor ID */
    0xYYYY, /* Product ID */
};
```

[Note]

1. Two TPLs are provided: one for the application program and the other for the hub (r_usb_hhubsys.c). If the TPL is set for a compliance test, do not set USB_NOVENDOR and USB_NOPRODUCT for the two TPLs. For how to set for compliance, see section 3.2.11, USB.

2. If the TPL is set for a compliance test, set the vendor ID and product ID of the hub used in the compliance test for the TPL for the hub.

11.8.7 **API Functions**

The MGR, HUBCD and HDCD request control of the hardware via API functions.

ANSI C99 is used. The types are defined in stdint.h.

The APIs in this driver follow the naming conventions for Renesas APIs.

All API calls and the interface definitions supporting these API calls are described in r_usb_basic_if.h.

Table 3-73 lists the API functions.

11.8.8 Callback Functions

Using callback functions eliminates the need to poll for events.

For example, when a user application requests data transfer by the HCD, a callback function is registered as a way of notifying the application of the result of data transfer to or from the device.

The tables below list the callback functions.

Table 11-7 R_usb_hstd_TransferStart Callback

Name	Callback function when requesting data transfer		
Function call syntax	void (*USB_UTR_CB_t)(USB_UTR_t*)		
Arguments	USB_UTR_t*		Pointer to USB_UTR_t as an argument of the R_usb_hstd_TransferStart() function
Returned values	—	–	–
Description	This function is called at the end of data transfer (i.e., the end of transfer of data of the size specified by the application, or the end of transfer when a short packet is received, etc.).		
Remarks			

Table 11-8 classcheck Callback

Name	classcheck callback function		
Function call syntax	void (*USB_CB_CHECK_t)(uint16_t **)		
Arguments	uint16_t	**table	table[0] Start address of the device descriptor table[1] Start address of the configuration descriptor table[2] Start address of the interface descriptor table[3] Descriptor check result table[4] HUB spec table[5] Port number (not used) table[6] Communication rate (not used) table[7] Device address
Returned values	—	–	–
Description	If the class code (member *ifclass* of the USB_HCDREG_tstructure) registered in the registration procedure matches that in the received descriptor, the function registered in member *classcheck* in the USB_HCDREG_t structure is called. table[3] Check result Respond to indicate whether the HDCD is ready to operate by returning either of the following: USB_OK: HDCD is ready to operate. USB_ERROR: HDCD is not ready to operate. table[4]: HUB spec (for the hub driver) USB_FSHUB: For a full-speed hub USB_HSHUBS: For a high-speed hub (single) USB_HSHUBM: For a high-speed hub (multiple) table[7] Device address Device address which is assigned by the host at the set address		
Remarks			

Table 11-9 State Change Callback

Name	State change callback function		
Function call syntax	void (*USB_CB_t)(uint16_t)		
Arguments	uint16_t	devaddr	Address of the connected device
Returned values	—	–	–
Description	If a state change occurs, the function registered in a member of the USB_HCDREG_t structure in the registration procedure is called. devconfig: Issued when the Set_Configuration request is issued. devdetach: Executed when detachment is detected. devsuspend: Executed when the state changes to “Suspended”. devresume: Executed when the state changes to “Resume”.		
Remarks			

11.8.9 USB Communications

(1) Pipes

A pipe is a logical connection to an endpoint on a USB device.

A default pipe for control transfer (pipe number 0) is set in the USB-BASIC firmware.

For data transfer, pipe information must be set from the descriptors of the USB device by calling the API function R_usb_hstd_SetPipe() at the time of class checking. Pipe information is allocated sequentially from pipe number 1 for each endpoint descriptor.

The maximum number of pipes can be changed by editing the config file. See section 3.2.11, USB.

The information to be set for the pipes is as follows.

- Interface number
- Endpoint number
- Endpoint transfer type
- Endpoint direction
- Maximum packet size
- Device address

A pipe number must be specified to request data transfer.
A pipe number can be obtained by calling API function R_usb_hstd_GetPipeID().
If the device is disconnected, pipe information must be cleared to release the pipe information area.
Pipe information can be cleared by calling API function R_usb_hstd_ClearPipe().

(2) Data transfer requests

A data transfer request can be issued by transferring the USB_UTR_t structure in which transfer information has been set as an argument of the API function R_usb_hstd_TransferStart().

For details of the USB_UTR_t structure, see section Table 3-21 USB_UTR_t Structure.

(3) Conveying the result of transfer

When data transfer is completed, the USB-BASIC firmware notifies the HDCD of the end of data transfer by using the callback function registered when a data transfer request has been issued.
The results of transfer are stored in member *result* of the USB_UTR_t structure.
The results of transfer are listed below.
USB_CTRL_END: Normal end of control transfer
USB_DATA_NONE: Normal end of data transmission
USB_DATA_OK: Normal end of data reception
USB_DATA_SHT: Normal end of data reception with the data length less than the specified value
USB_DATA_OVR: Received data size exceeded
USB_DATA_ERR: Non-response condition or overrun/underrun error detected
USB_DATA_STALL: STALL response or MaxPacketSize error detected
USB_DATA_STOP: Forced end of data transfer

(4) Retrying transfer

When a no-response condition occurs on a pipe, the USB-BASIC firmware performs communication retries.
If the no-response condition is detected, USB_DATA_ERR is returned to the HDCD.

(5) **Notes when receiving a short packet**

When a short packet is received, the expected remaining receive data length is stored in *tranlen* of the USB_UTR_t structure and transfer ends. When the received data is larger than the buffer size, data is read from the FIFO buffer up to the buffer size and transfer ends.

(6) **Control transfer**

When the USB-BASIC firmware is used to transmit requests (vendor or class requests) not listed in the USB standard requests in section 11.8.4, Host Manager (MGR), to a device, the following must be set in the USB_UTR_t structure member before R_usb_hstd_TransferStart() is called.

- keyword: Pipe number (USB_PIPE0)
- tranadr: Data buffer (data stage)
- tranlen: Transfer size
- setup: Setup data
- complete: Callback function

The callback function is set to *complete* on completion of data transfer and conveys the result of transfer (see section 7.13.2, Notification of Transfer Result).
The following is an example of transmitting vendor requests.

```
<Vendor request transmission example>
uint8_t usb_gsmp_VendorRequestData[16];
USB_SETUP_t usb_gsmp_VendorRequest;
USB_UTR_t usb_gsmp_ControlUtr;
USB_UTR_CB_t usb_gsmp_VendorRequestCb;

USB_ER_t usb_hsmp_VendorRequestProcess(void)
{
    USBC_ER_t err;

    /* Set setup data */
    usb_gsmp_VendorRequest.type = ((bRequest << 8) | bmRequestType);
    usb_gsmp_VendorRequest.value = wValue;
    usb_gsmp_VendorRequest.index = wIndex;
    usb_gsmp_VendorRequest.length = wLength;
    usb_gsmp_VendorRequest.devaddr = devaddr;

    /* Set the transfer pipe (default pipe) */
    usb_gsmp_ControlUtr.keyword = USB_PIPE0;
    /* Set data for transmission */
    usb_gsmp_ControlUtr.tranadr = usb_gsmp_VendorRequestData;
    /* Set the transfer size */
    usb_gsmp_ControlUtr.tranlen = usb_gsmp_VendorRequest.length;
    /* Set the Setup command */
    usb_gsmp_ControlUtr.setup = &usb_gsmp_VendorRequest;
    /* Set the callback function */
    usb_gsmp_ControlUtr.complete = &usb_gsmp_VendorRequestCb;

    /* Request data transmission */
    err = R_usb_hstd_TransferStart(&usb_gsmp_ControlUtr);

    return err;
}
```

(7) **Data transfer**

When the USB-BASIC firmware is used, the following must be set in the USB_UTR_t structure before R_usb_hstd_TransferStart() is called.

- keyword: Pipe number
- tranadr : Data buffer
- tranlen: Transfer size
- complete: Callback function

The callback function is set to *complete* on completion of data transfer and conveys the result of transfer (see section 7.13.2, Notification of Transfer Result).

The following is an example of bulk-in transfer.

<Bulk-in transfer example>

```
uint8_t usb_gsmp_VendorBulkInData[512];
USB_UTR_t usb_gsmp_DataUtr;
USB_UTR_CB_t usb_gsmp_VendorBulkInCb;


USB_ER_t usb_hsmp_VendorBulkInProcess(uint16_t devaddr)
{
    USB_ER_t err;

    /* Set the transfer pipe */
    usb_gsmp_DataUtr.keyword = R_usb_htsd_GetPipeID(devaddr, USB_EP_BULK, USB_EP_IN, 0);
    /* Set data for transmission */
    usb_gsmp_DataUtr.tranadr = usb_gsmp_VendorBulkInData;
    /* Set the transfer size */
    usb_gsmp_DataUtr.tranlen = 512;
    /* Set the callback function */
    usb_gsmp_DataUtr.complete = &usb_gsmp_VendorBulkInCb;

    /* Request data transmission */
    err = R_usb_hstd_TransferStart(&usb_gsmp_DataUtr);

    return err;
}
```

11.8.10 Non-OS Scheduler

(1) Overview

The non-OS scheduler carries out task scheduling according to the priority of tasks.

The non-OS scheduler has the following features:

- Manages requests generated by tasks and hardware according to their relative priority.
- When multiple task requests with the same priority are generated, a FIFO queue determines their order of execution.
- Callback functions are used for responses to tasks indicating the end of a request, so the user need only install appropriate class drivers for the system and there is no need to modify the scheduler itself.

(2) Non-OS scheduler macros

The table below lists the scheduler macros available to the user.

Table 11-10 Scheduler Macros

Scheduler Macro	Registration Function	Description
R_USB_SND_MSG	R_usb_cstd_SndMsg	Sends a message by specifying the message box.
R_USB_WAI_MSG	R_usb_cstd_WaiMsg	Executes USB_SND_MSG after executing the scheduler specified number of times
R_USB_RCV_MSG	R_usb_cstd_RecMsg	Checks the specified mailbox if any message is received.
R_USB_PGET_BLK	R_usb_cstd_PgetBlk	Secures the area for storing the message.
R_USB_REL_BLK	R_usb_cstd_RelBlk	Releases the area for storing the message.

(3) Non-OS scheduler API functions

The API functions used by the non-OS scheduler are listed in

Table 3-74.

11.9 Host Mass Storage Class Driver (HMSC)

The HMSC consists of the FSI, HMSDD and HMSCD, and is combined with the USB-BASIC firmware.

This section describes the functions of the HMSC.

11.9.1 Class Requests

The table below lists the class requests that the HMSC supports.

Table 11-11 Class Requests

Request	Code	Description	Support
Mass Storage Reset	0xFF	Clears the protocol error.	Yes
GetMaxLUN	0xFE	Acquires the maximum number of units that the device supports.	Yes

Yes: Implemented, No: Not implemented

11.9.2 Storage Commands

The table below lists the storage commands that the HMSC supports.

Table 11-12 Storage Commands

Command Name	Code	Description	Support
TEST_UNIT_READY	0x00	Checks the state of the peripheral device.	Yes
REQUEST_SENSE	0x03	Acquires the state of the peripheral device.	Yes
FORMAT_UNIT	0x04	Format of the logical unit	No
INQUIRY	0x12	Acquires parameter information of the logical unit.	Yes
MODE_SELECT6	0x15	Specifies parameters.	Yes
MODE_SENSE6	0x1A	Acquires parameters of the logical unit.	No
START_STOP_UNIT	0x1B	Enables or disables access to the logical unit.	No
PREVENT_ALLOW	0x1E	Enables or disables taking out the media.	Yes
READ_FORMAT_CAPACITY	0x23	Acquires the formattable capacity.	Yes
READ_CAPACITY	0x25	Acquires capacity information of the logical unit.	Yes
READ10	0x28	Reads data.	Yes
WRITE10	0x2A	Writes data.	Yes
SEEK	0x2B	Moves to the logical block address.	No
WRITE_AND_VERIFY	0x2E	Writes and verifies data.	No
VERIFY10	0x2F	Verifies data.	No
MODE_SELECT10	0x55	Specifies parameters.	No
MODE_SENSE10	0x5A	Acquires parameters of the logical unit.	Yes

Yes: Implemented, No: Not implemented

11.9.3 Checking the USB Storage Devices

The USB-BASIC firmware uses a callback function to notify the HMSC of the information obtained via the GET_DESCRIPTOR request during enumeration. The HMSCD has the API function R_usb_hmsc_ClassCheck() to be registered as this callback function.

R_usb_hmsc_ClassCheck() analyzes the descriptor information and notifies the check result by the API function R_usb_hstd_ReturnEnuMGR() of the USB-BASIC firmware. If the result has no problem, the USB-BASIC firmware finishes enumeration.

11.9.4 Acquiring Information on the USB Storage Devices

After enumeration is finished, information on the USB storage devices can be obtained by calling the API function R_usb_hmsc_StrgDriveSearch() of the HMSDD. The completion of this processing is notified by the callback function registered when calling R_usb_hmsc_StrgDriveSearch().

The HMSC runs on the assumption that the number of units is 0 irrespective of the response to the GetMaxLUN request. Therefore, only LUN0 is supported.

The figure below shows the sequence of acquiring information on the USB storage devices.

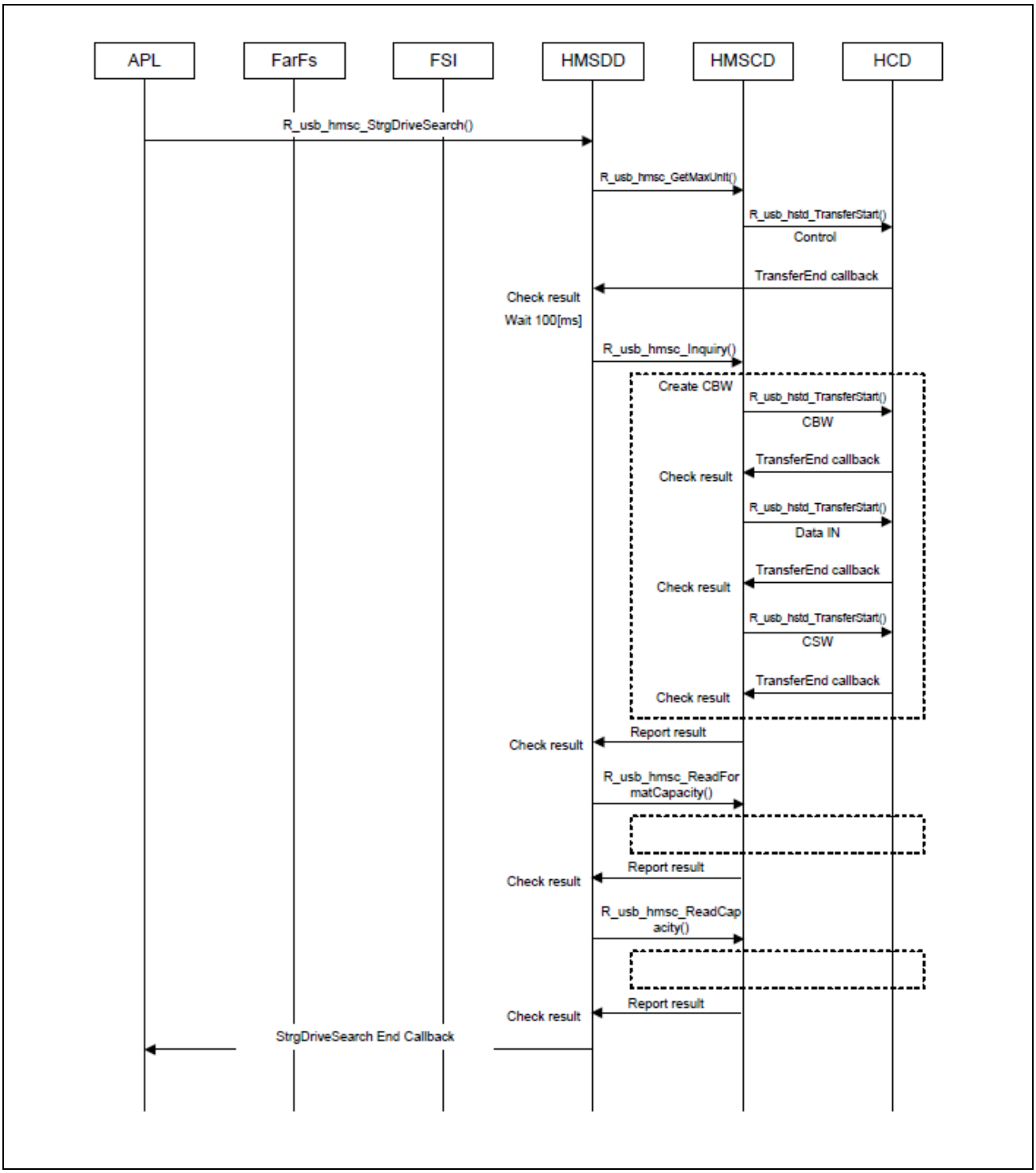


Figure 11-6 Sequence of Acquiring Information on the USB Storage Devices

11.9.5 Access to the USB Storage Devices

After acquiring information, the HMSC can access the USB storage devices by an API function of FatFs. When an API function of FatFs is called, the FSI is called while the file system is being processed and the HMSDD runs.
The HMSDD calls the API function of the HMSCD as required by the processing.
The HMSDC that runs accordingly issues a class request and creates USB packets according to the BOT protocol.

The BOT protocol performs data transfer according to the LBA and specifies the transfer size by the number of bytes.
The figure below shows the sequence of accessing the USB storage devices.

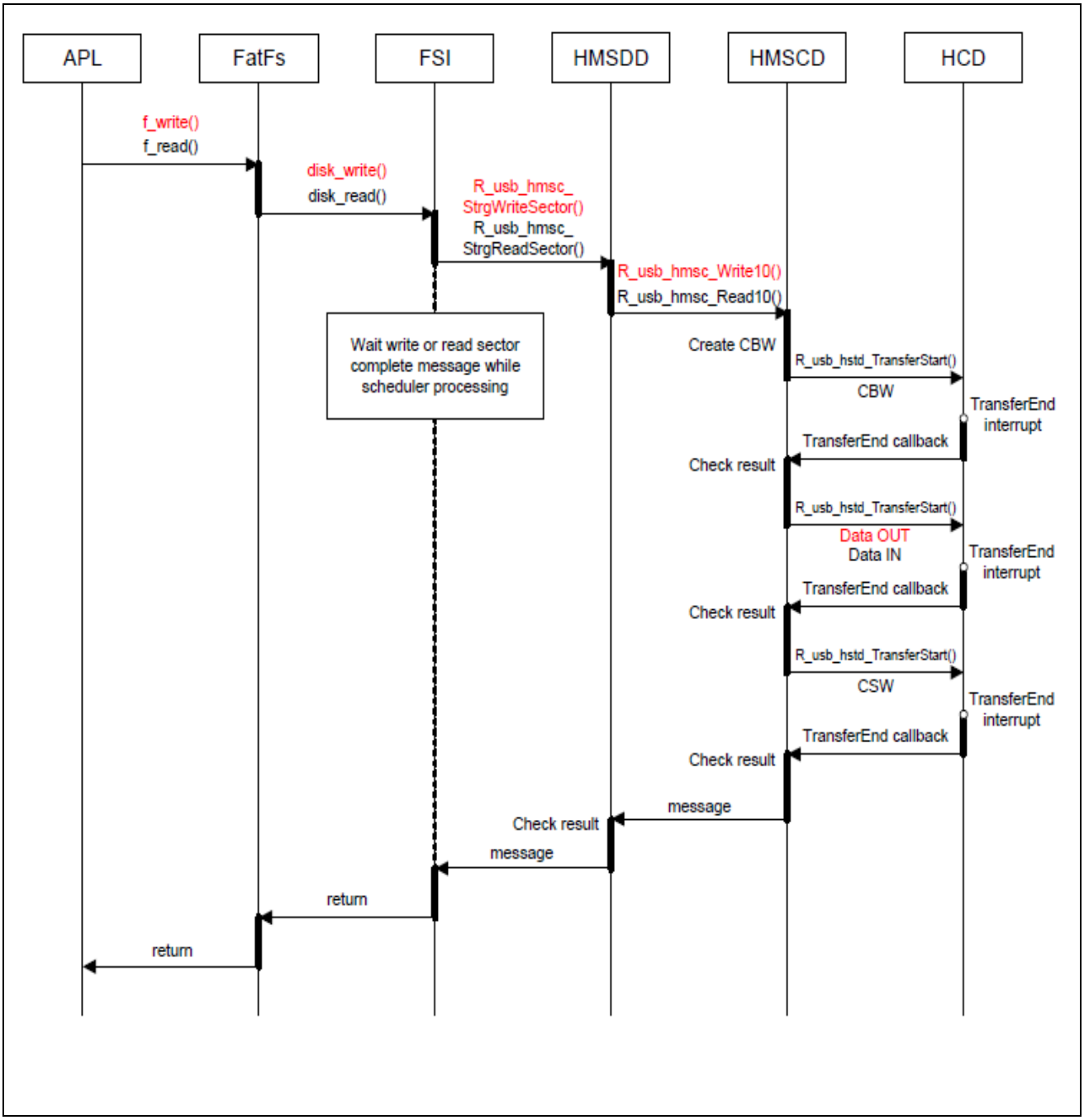


Figure 11-7 Sequence of Access to the USB Storage Devices

11.9.6 API Functions of the HMSCD

HMSCD has API functions mainly for the HMSDD.

Usually three functions are used by the application: R_usb_hmsc_Task(), R_usb_hmsc_driver_start() and R_usb_hmsc_Class_Check().

The API functions of the HMSCD are listed in Table 3-75.

11.9.7 API Functions of the HMSDD

The API functions of the HMSDD are listed in Table 3-76.

11.9.8 FSI Functions

FatFs is simply a file system layer and does not include the storage device control layer.

FatFs requests the lower layer of an interface, so control functions must be provided for the platform and storage device used.

The HMSC has sample control functions (FSI functions). Check the FatFs specifications and modify the functions for each system if required.

The FSI functions are listed in Table 3-77.

11.9.9 Setting the Scheduler

The table below lists the HMSC scheduler settings.

Table 11-13 Scheduler Settings

Function Name	Task ID	Priority	Mailbox Name	Memory Pool	Description
R_usb_hmsc_StrgDriveTask	USB_HSTRG_TSK	USB_PRI_3	USB_HSTRG_MBX	USB_HSTRG_MPL	HSTRG task
R_usb_hmsc_task	USB_HMSC_TSK	USB_PRI_3	USB_HMSC_MBX	USB_HMSC_MPL	HMSCD 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

12. Website and Support

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

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

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

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	May. 10, 2017	-	First Edition issued
1.10	Sep. 1, 2017	373	Added WDTA Sample Software
		392	Added IWDTA Sample Software
		409	Added RIIC Sample Software
		432	Added USB Host Sample Software
1.20	Sep. 17,2018	2. File structure	
		21	Table 2-4 Modify the file structure for the E2studio Ver6 support. Delete CDC_Demo_Win7.inf from usbf / pcdc / utilities /
		24	Table 2-5 Modify the file structure for the E2studio Ver6 support. Added EC-1_init_serial_boot.gsi to usbh_sample / GCC
		25	Table 2-6 Modify the file structure for the E2studio Ver6 support. EC-1_init_serial_boot.gsi added to Templates / GCC / serial_boot Added usrheap.c to Templates / GCC
		3. Driver	
		30	Table 3-15 structure members added
		40	Table 3-47 Transmit / receive FIFO buffer stage number 128 Message setting deleted.
		44	Table 3-50 Corrected omission of constants
		59	Table 3-66 Modfy function name. Add an explanation. Function addition.
		112~125	3.7 Modfy, RSPI Control,Function name
		113	3.7.2 Added R_RSPIm_Pin_Init function.
		117	3.7.6 Added R_RSPIm_ClearState function.
		4. CAN Sample soft	
		249	Table 4-3 Adding Local Functions
		251	4.4.1 Modify the main return value (void → int)
		252	4.4.2 Added can_main_init function
		253	4.4.3 Added ecm_init function
		265	4.4.4 Added icu_init function
		302	Figure 4-2 Corrected setting example of interrupt and callback function in figure.
		303	Figure 4-3 Corrected setting example of interrupt and callback function in figure.
		307	Figure 4-6 Corrected setting example of interrupt and callback function in figure.
		308	Figure 4-7 Corrected setting example of interrupt and callback function in figure.
		309	Figure 4-8 Corrected setting example of interrupt and callback function in figure.
		314	Figure 4-12 Corrected setting example of interrupt and callback function in figure. Change the title

	317	Figure 4-15 Corrected setting example of interrupt and callback function in figure. Change the title
	318	Figure 4-16 Change the title
	319	Figure 4-17 Corrected setting example of interrupt and callback function in figure. Change the title
	320	Figure 4-18 Change the title
	321	Figure 4-19 Corrected setting example of interrupt and callback function in figure. Change the title
	322	Figure 4-20 Change the title
	323	Figure 4-21 Corrected setting example of interrupt and callback function in figure. Change the title
	324	Figure 4-22 Change the title
	325~332	4.5.6 Callback processing Delete description related to CAN 0
	333	Table 4-4 Interrupt Sources Delete the description about CAN 0, correct the error of interrupt priority
	334	4.6.3 Preparation for use (transmission test / reception test) Corrected error in terminal number
5. RSPI Sample software		
	344	5.1 Overview Changed used channel from 1 to 0
	355	Figure 5-1 Change the title
	356	Figure 5-2 Change the title
	357	Table 5-2 Changed used channel from 1 to 0
	358	Figure 5-4 Modify the setting of the channel 0
	359	5.5.5 modify the sample program execution Example Description
7. USB function sample software		
	391	Table 7-4 Add the EVENT_DETACH to the table
	392	Table 7-6 Modify the scope of port_init to global
	394	7.5.2 remove the static
11. USB host sample software		
	504	Table 11-4 soft_wait function is unnecessary and removed
	508	soft_wait function delete
	536	Modify the version of the Fat module (R0.11 → R0.13)

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.

- Arm and Cortex are registered trademarks of Arm Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.
- Ethernet is a registered trademark of Fuji Xerox Co., Ltd.
- IEEE is a registered trademark of the Institute of Electrical and Electronics Engineers Inc
- TRON is an acronym for "The Real-time Operation system Nucleus.
- ITRON is an acronym for "Industrial TRON.
- μITRON is an acronym for "Micro Industrial TRON.
- TRON, ITRON, and μITRON do not refer to any specific product or products.
- EtherCAT® and TwinCAT® are registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany.
- Additionally all product names and service names in this document are a trademark or a registered trademark which belongs to the respective owners.

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.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
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 product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

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.
1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.
Tel: +1-408-432-8888, Fax: +1-408-434-5351

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: +44-1628-651-700, Fax: +44-1628-651-804

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, 200333 P. R. China
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 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338

© 2018 Renesas Electronics Corporation. All rights reserved.

Colophon 7.0