RX Family

High-Speed I²C Bus Interface (RIICHS) Module Using Firmware Integration Technology

Introduction

This application note describes the High-Speed I²C bus interface (RIICHS) module using firmware integration technology (FIT) for communications between devices using the I²C bus interface.

Target Device

- RX671 Group

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

Target Compilers

- Renesas Electronics C/C++ Compiler Package for RX Family
- GCC for Renesas RX
- IAR C/C++ Compiler for Renesas RX

Contents

1. Overview

The High Speed I²C bus interface module using firmware integration technology (RIICHS FIT module (1)) provides a method to transmit and receive data between the master and slave devices using the I²C bus interface (RIICHS). The RIICHS is in compliance with the NXP I²C-bus (Inter-IC-Bus) interface.

Note:
1. When the description says “module” in this document, it indicates the RIICHS FIT module.

Features supported by this module are as follows:

- Master transmission, master reception, slave transmission, and slave reception

- Multi-master configuration that communicates between multiple masters and one slave.

- Communication mode can be standard mode (maximum communication rate is 100 kbps) and fast mode (maximum communication rate is 400 kbps).
- Channel 0 of RX671 supports fast mode plus (maximum communication rate is 1 Mbps) and high speed mode (maximum communication rate is 3.4 Mbps).

Limitations

This module has the following limitations:

(1) The module cannot be used with the DMAC and the DTC.

(2) The NACK arbitration-lost detection function of the RIICHS is not supported.

(3) Transmission with 10-bit address is not supported.

(4) Acceptance of the restart condition on slave device mode is not supported. Do not specify the address of a device in which this module is embedded as an address immediately following a restart condition.

(5) The module does not support multiple interrupts.

(6) API function calls except for the R_RIICHS_GetStatus function is prohibited within a callback function.

(7) Set the I flag to 1 to use interrupts.

1.1 RIICHS FIT Module

This module is implemented in a project and used as the API. Refer to 2.12 Adding the FIT Module to Your Project for details on implementing the module to the project.

1.2 Outline of the API

Table 1.1 lists the API Functions.

Table 1.1 API Functions

Item	Contents
R_RIICHS_Open()	The function initializes the RIICHS FIT module. This function must be called before calling any other API functions.
R_RIICHS_MasterSend()	Starts master transmission. Changes the master transmit pattern according to the parameters. Operates batched processing until stop condition generation.
R_RIICHS_MasterReceive()	Starts master reception. Changes the master receive pattern according to the parameters. Operates batched processing until stop condition generation.
R_RIICHS_SlaveTransfer()	Performs slave transmission and reception. Changes the transmit and receive patterns according to the parameters.
R_RIICHS_GetStatus()	Returns the state of this module.
R_RIICHS_Control()	This function outputs conditions, Hi-Z from the SDA pin, and one-shot of the SCL clock. Also it resets the settings of this module. This function is mainly used when a communication error occurs.
R_RIICHS_Close()	This function completes the RIICHS communication and releases the RIICHS used.
R_RIICHS_GetVersion()	Returns the current version of this module.

1.3 Overview of RIICHS FIT Module

1.3.1 Specifications of RIICHS FIT Module

1. This module supports master transmission, master reception, slave transmission, and slave reception.

   - There are four transmit patterns that can be used for master transmission. Refer to 1.3.2 for details on master transmission.

   - Master reception and master transmit/receive can be selected for master reception. Refer to 1.3.3 for details on master reception.

   - Slave reception or slave transmission is performed according to the content of the data transmitted from the master. Refer to 1.3.4 for details on slave reception and slave transmission.

2. An interrupt occurs when any of the following operations completes: start condition generation, slave address transmission/reception, data transmission/reception, NACK detection, arbitration-lost detection, or stop condition generation. In the RIICHS interrupt handling, the communication control function is called and the operation is continued.

3. Multiple slave devices with different addresses on the channel bus can be controlled. However, while communication is in progress (the period from start condition generation to stop condition generation), communication with other devices is not available. Figure 1.1 shows an Example of Controlling Multiple Slave Devices.

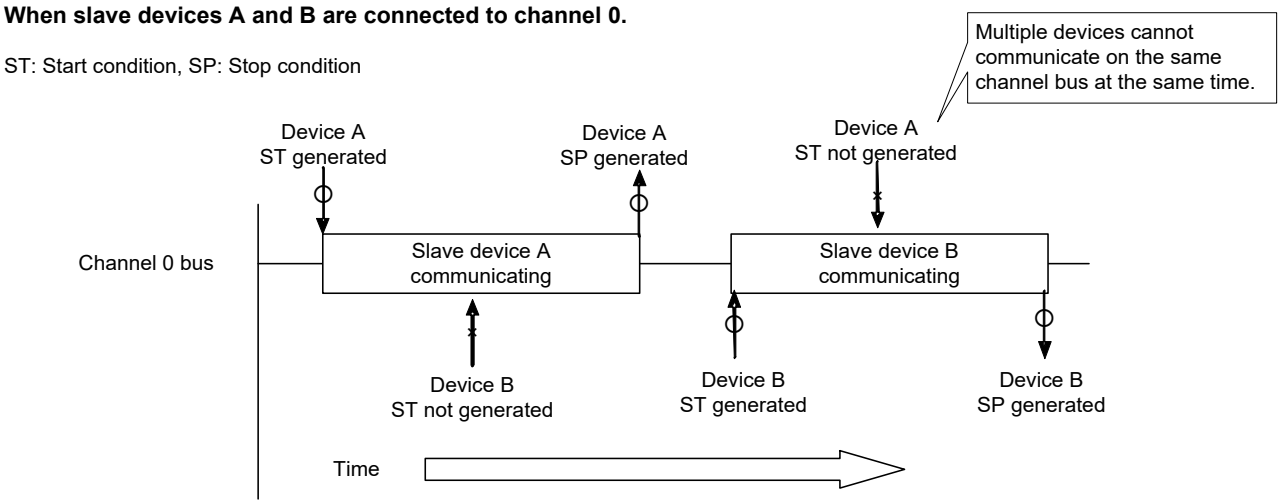


Figure 1.1 Example of Controlling Multiple Slave Devices

1.3.2 Master Transmission

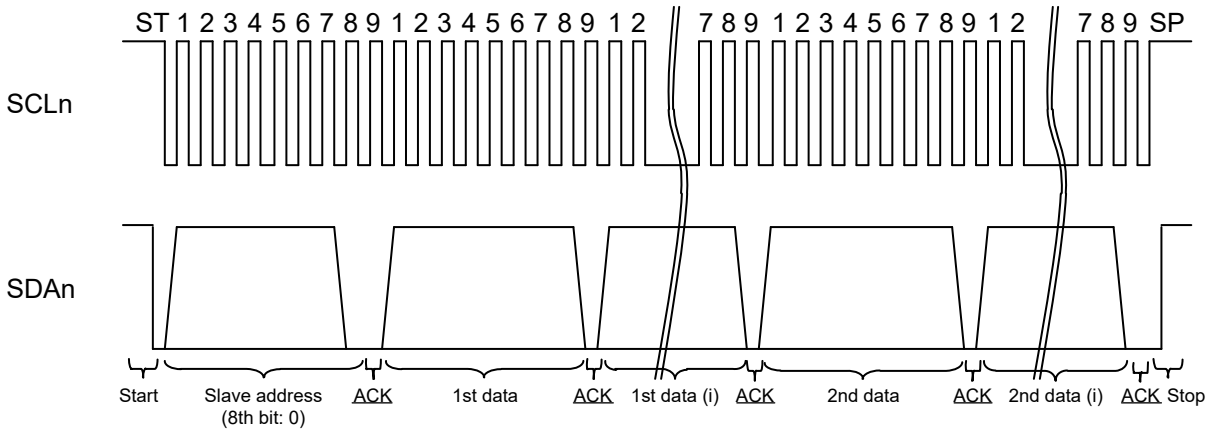
The master device (master (RX MCU)) transmits data to the slave device (slave).

With this module, four patterns of waveforms can be generated for master transmission. A pattern is selected according to the arguments set in the parameters which are members of the I²C communication information structure. Figure 1.2 to Figure 1.4 Signals for Pattern 3 of Master Transmission show the transmit patterns. Refer to 2.9 Parameters for details on the I²C communication information structure.

(1) Pattern 1

The master (RX MCU) transmits data in two buffers for the first data and second data to the slave.

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



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

Figure 1.2 Signals for Pattern 1 of Master Transmission

(2) Pattern 2

The master (RX MCU) transmits data in the buffer for the second data to the slave.

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

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

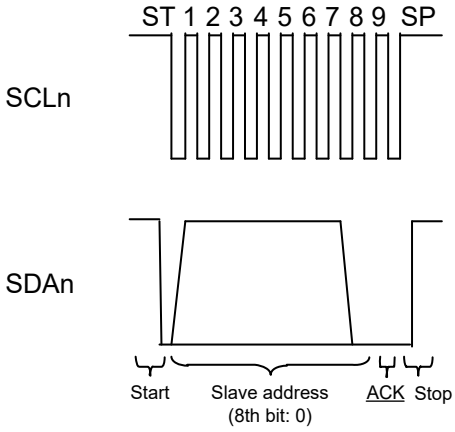
Figure 1.3 Signals for Pattern 2 of Master Transmission

(3) Pattern 3

The master (RX MCU) transmits only the slave address to the slave.

Operations from start condition generation through to slave address transmission are the same as the operations for pattern 1. After transmitting the slave address, if neither the first data nor the second data are set, data transmission is not performed, then a stop condition 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.



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

Figure 1.4 Signals for Pattern 3 of Master Transmission

Figure 1.5 shows the procedure of master transmission. The callback function is called after generating a stop condition. Specify the function name in the CallBackFunc of the I²C communication information structure member.

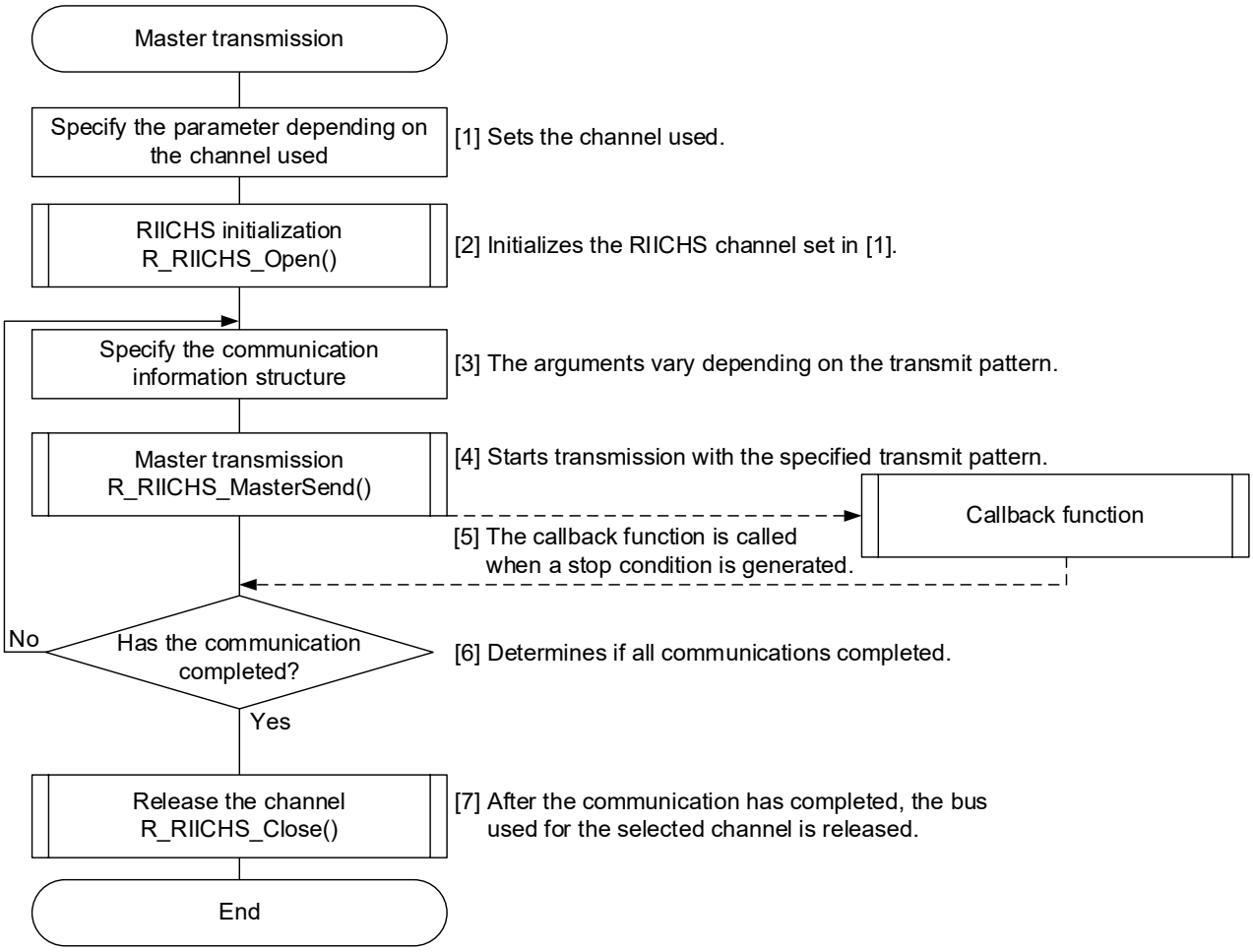


Figure 1.5 Example of Master Transmission

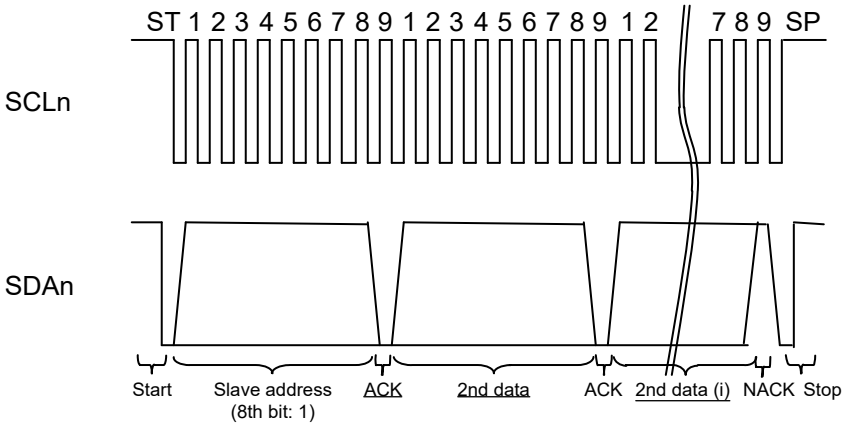
1.3.3 Master Reception

The master (RX MCU) receives data from the slave. This module supports master reception and master transmit/receive. The receive pattern is selected according to the arguments set in the parameters which are members of the I²C communication information structure. Figure 1.6 and Figure 1.7 show receive patterns. Refer to 2.9 Parameters for details on the I²C communication information structure.

(1) Master Reception

The master (RX MCU) receives data from the slave.

A start condition is generated and then the slave address is transmitted. The eighth bit specifies the transfer direction. This bit is set to 1 (read) when receiving. Then data reception starts. An ACK is transmitted each time 1-byte data is received except the last data. A NACK is transmitted when the last data is received to notify the slave that all data receptions have completed. Then a stop condition is generated and the bus is released.



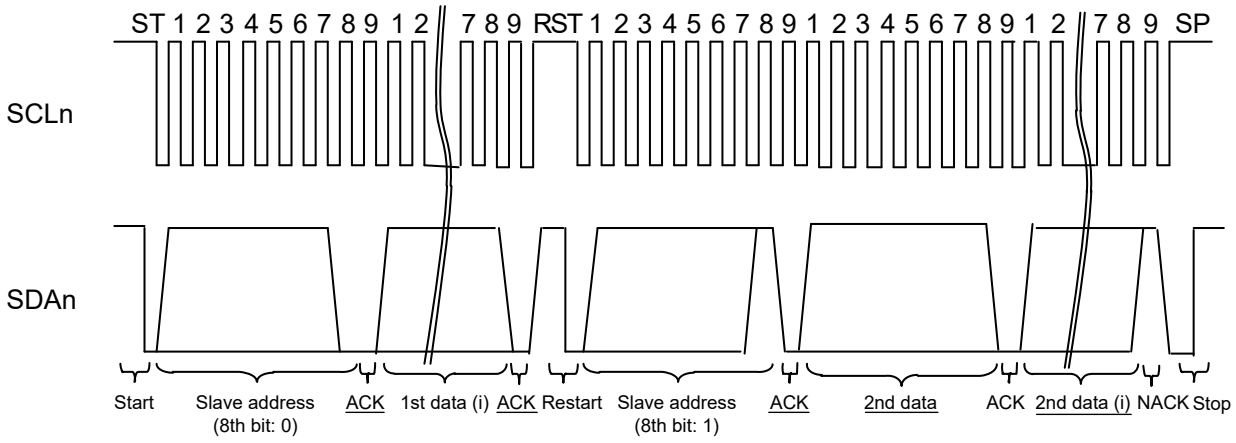
n: Channel number
ST: Start condition generation NACK: Acknowledge: 1
SP: Stop condition generation ACK: Acknowledge: 0
* A signal with an underline indicates data transmission from the slave to the master.

Figure 1.6 Signals for Master Reception

(2) Master Transmit/Receive

The master (RX MCU) transmits data to the slave. After the transmission completes, a restart condition is generated, and the master receives data from the slave.

A start condition is generated and then the slave address is transmitted. The eighth bit specifies the transfer direction. This bit is set to 0 (write) when transmitting. Then the first data is transmitted. When the data transmission completes, a restart condition 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 transmitted each time 1-byte data is received except the last data. A NACK is transmitted when the last data is received to notify the slave that all data receptions have completed. Then a stop condition is generated and the bus is released.



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

Figure 1.7 Signals for Master Transmit/Receive

Figure 1.8 shows the procedure of master reception. The callback function is called after generating a stop condition. Specify the function name in the CallBackFunc of the I²C communication information structure member.

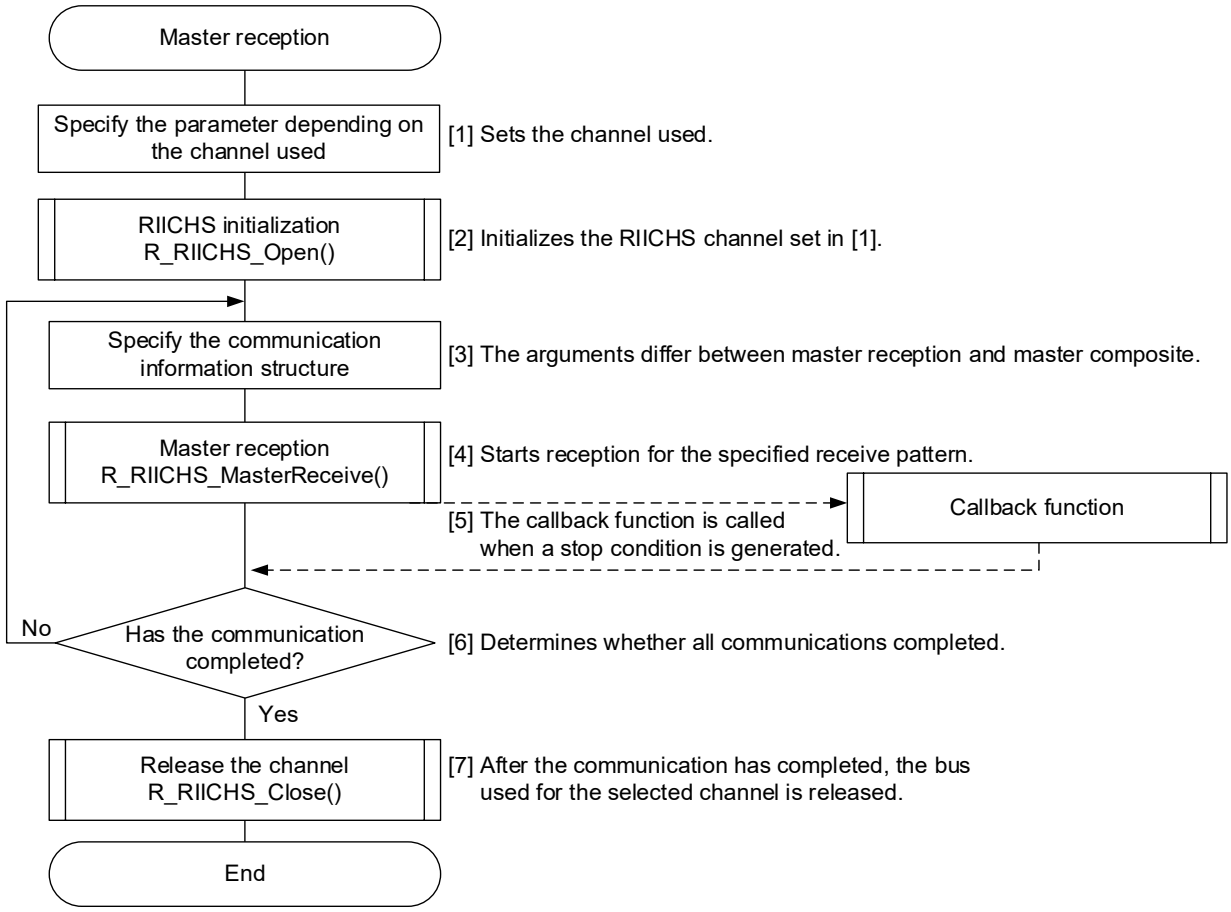


Figure 1.8 Example of Master Reception

1.3.4 Slave Transmission and Reception

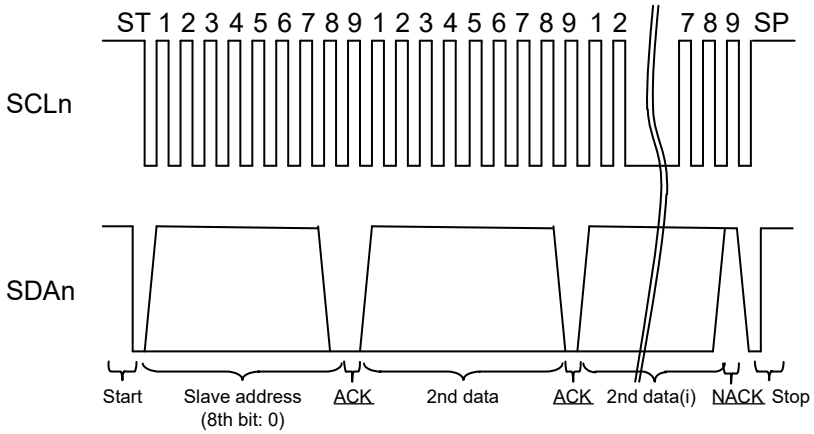
The slave (RX MCU) receives data transmitted from the master. The slave transmits data by the transmit request from the master.

When the slave address specified by the master matches the slave address set by argument when executing R_RIICHS_Open() function, slave transmission and reception starts. The module processes the operation automatically determining whether the operation is slave reception or slave transmission according to the eighth bit (transfer direction specify bit) of the slave address.

(1) Slave Reception

The slave (RX MCU) receives data from the master.

After a start condition generated by the master is detected, when the received slave address matches its own address and the eighth bit of the slave address is 0 (write), then the slave starts receive operation. When the last data (the number of data specified in the I²C communication information structure member) is received, a NACK is returned to the master to notify that all necessary data has been received. Figure 1.9 shows the Signals for Slave Reception.



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

Figure 1.9 Signals for Slave Reception

Figure 1.10 shows the procedure of slave reception. The callback function is called after generating a stop condition. Specify the function name in the CallBackFunc of the I²C communication information structure member.

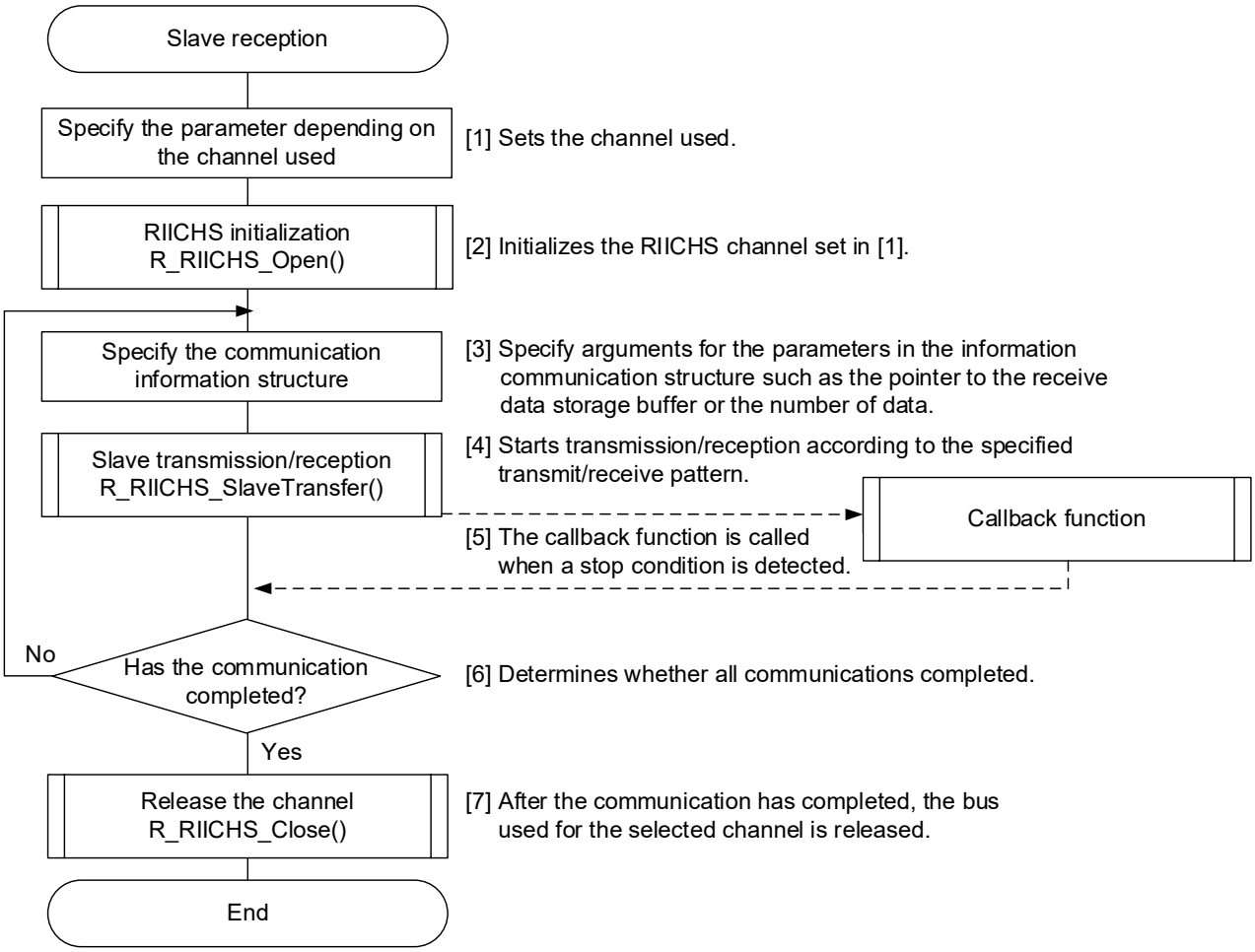
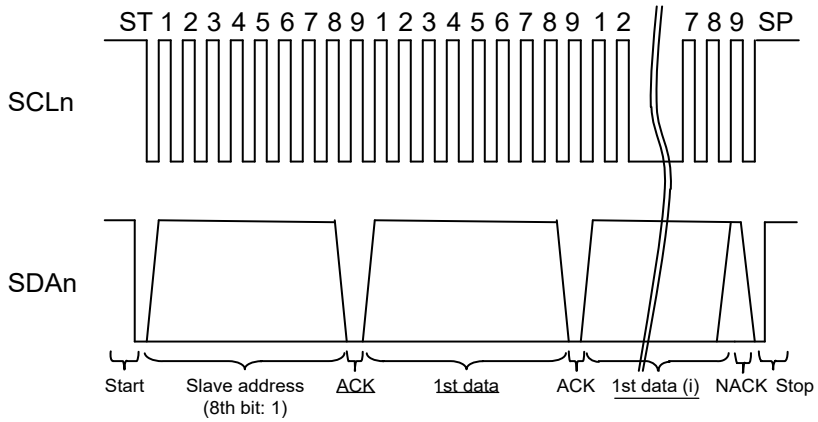


Figure 1.10 Slave Reception

(2) Slave Transmission

The slave (RX MCU) transmits data to the master.

After a start condition from the master is detected, when the slave address matches its own address and the eighth bit of the slave address is 1 (read), then the slave starts transmit operation. When the transmit request exceeds the number of data specified in the I²C communication information structure member, the slave transmits 0xFF as data. The slave continues transmit operation until a stop condition is detected. Figure 1.11 shows the Signals for Slave Transmission.



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

Figure 1.11 Signals for Slave Transmission

Figure 1.12 shows the procedure of slave transmission. The callback function is called after generating a stop condition. Specify the function name in the CallBackFunc of the I²C communication information structure member.

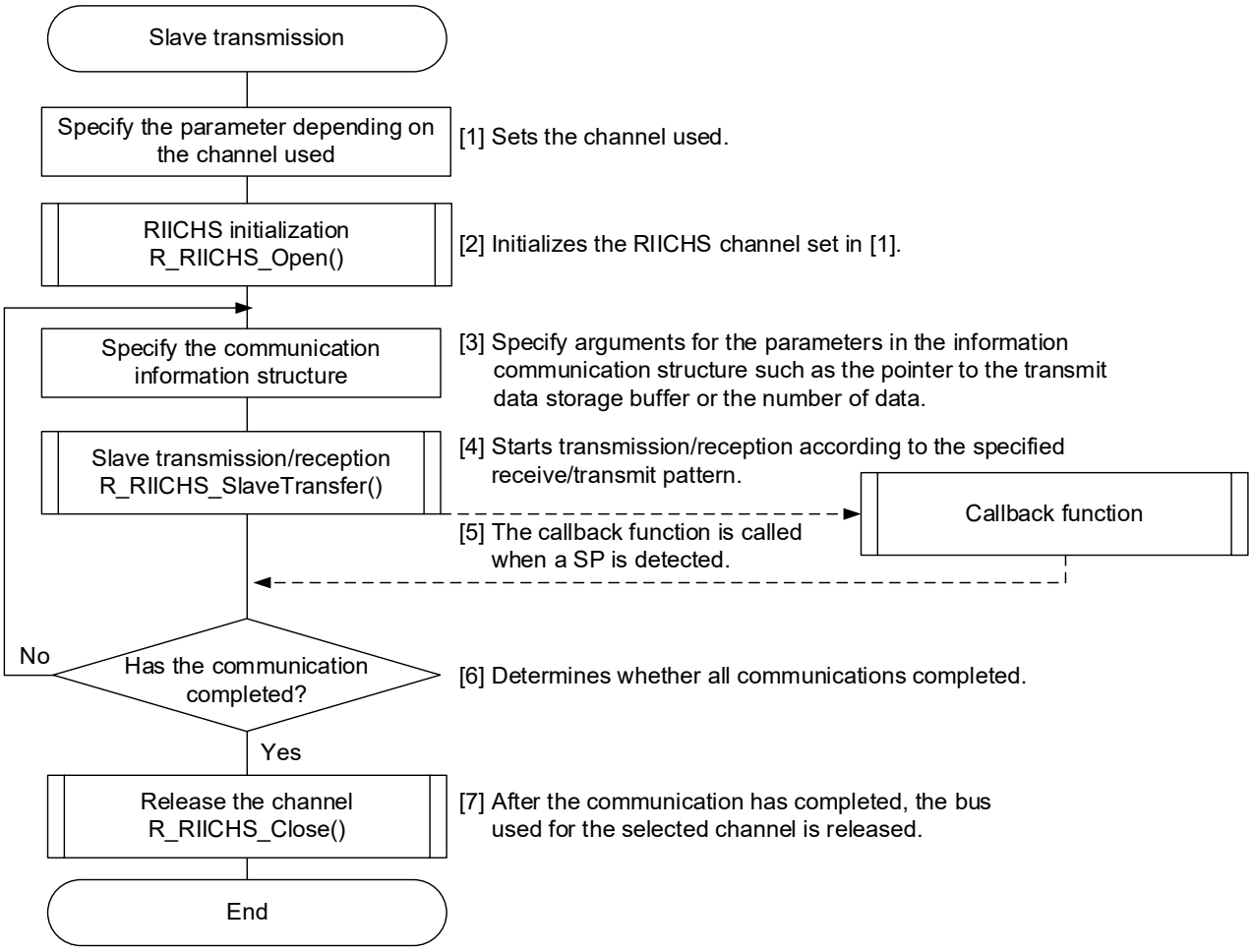


Figure 1.12 Slave Transmission

1.3.5 State Transition

Figure 1.13 shows the RIICHS FIT Module State Transition Diagram.

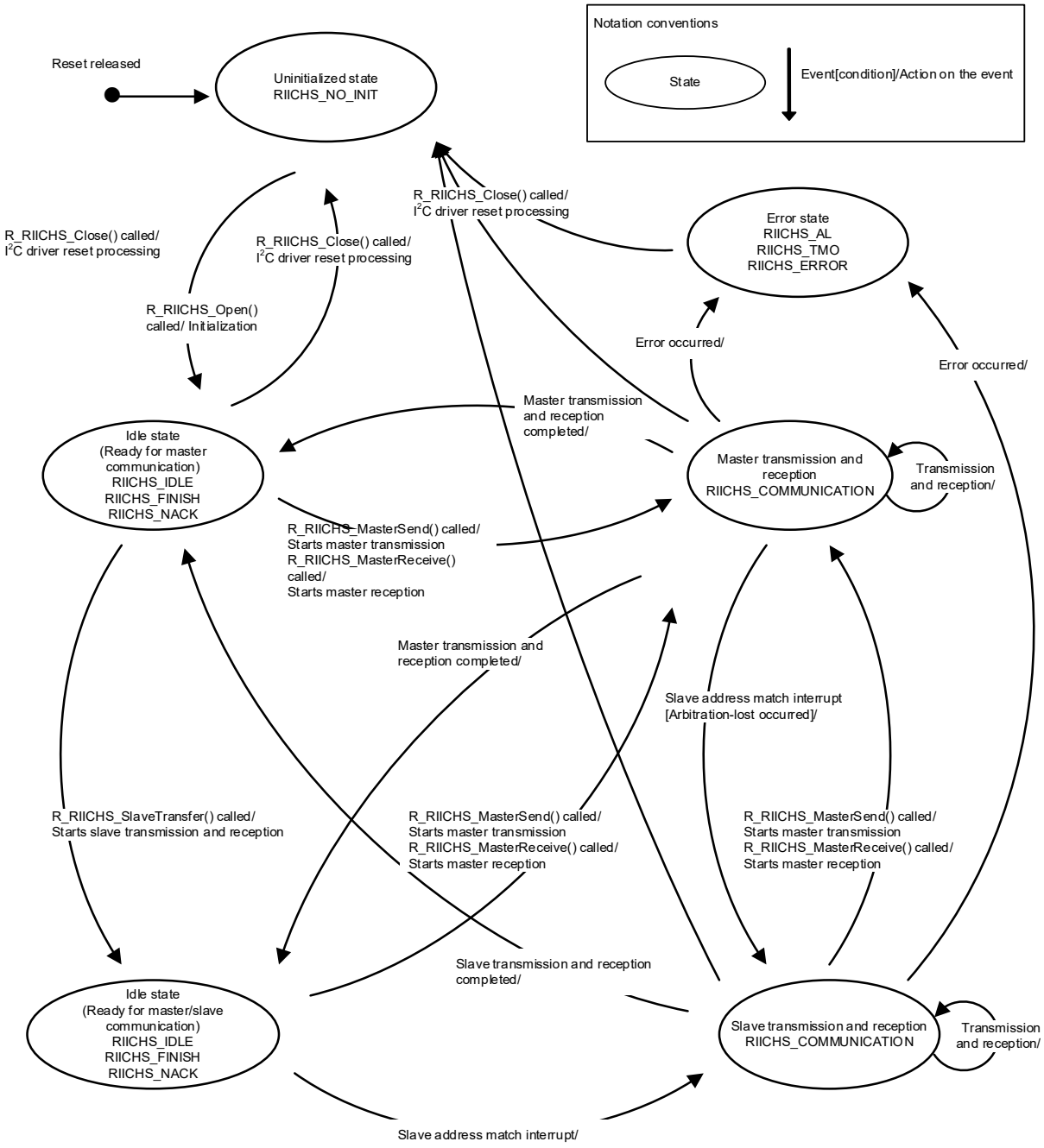


Figure 1.13 RIICHS FIT Module State Transition Diagram

1.3.6 Flags when Transitioning States

dev_sts is the device state flag and is one of the I²C communication information structure members. The flag stores the communication state of the device. Using this flag enables controlling multiple slaves on the channel.

Table 1.2 lists the Device State Flags when Transitioning States.

Table 1.2 Device State Flags when Transitioning States

State	Device State Flag (dev_sts)
Uninitialized state	RIICHS_NO_INIT
Idle states	RIICHS_IDLE RIICHS_FINISH RIICHS_NACK
Communicating (master transmission, master reception, slave transmission, and slave reception)	RIICHS_COMMUNICATION
Arbitration-lost detection state	RIICHS_AL
Timeout detection state	RIICHS_TMO
Error	RIICHS_ERROR

1.3.7 Arbitration-Lost Detection Function

This module detects arbitration-lost for the reasons below. The module does not support the arbitration-lost detection on slave transmission while the RIICHS does.

(1) When a start condition is issued during the bus busy state:

If the module issues a start condition when the other master has already issued a start condition and occupied the bus (bus busy state), the module detects arbitration-lost.

(2) When the module issues a start condition after the other master issued a start condition though the bus is free:

When the module issues a start condition, it attempts to drive the SDA line low. However if the other master issued a start condition earlier, the signal level on the SDA line does not match the signal level output by the module. Then the module detects arbitration-lost.

(3) When multiple start conditions are issued at the same time:

If multiple masters issue start conditions at the same time, the module may determine that the start condition has been issued successfully on each device. Then each device starts communication. However, when any of the conditions described below occurs, the module detects arbitration-lost.

a. When data transmitted by masters are different:

The module compares the signal level on the SDA line with the signal level output by itself during communication. If these signals do not match while data is being transmitted including the slave address, the module detects arbitration-lost.

b. The numbers of data transmissions differ between masters while data sent by the masters are the same.

With the case other than the above a, i.e., the slave address and transmit data match, the module does not detect arbitration-lost. However if the number of data transmitted by masters differ, the module detects arbitration-lost.

1.3.8 Timeout Detection Function

The timeout detection function can be enabled in this module (enabled as default). The RIICHS can detect an abnormal bus state by monitoring that the SCL0 line is stuck low or high for a predetermined time.

The timeout detection function detects a bus hang up, i.e. the SCL line is held low or high, in the following period:

(1) The bus is busy in master mode.

(2) The RIICHS's own slave address is detected and the bus is busy in slave mode.

(3) The bus is free while generation of a START condition is requested.

Refer to "2.9 Parameters" for details on enabling and disabling the timeout detection function.

Refer to 6.3 Timeout Detection and Processing After the Detection for detailed explanation when a timeout is detected.

2. API Information

The FIT module provided with this application note has been confirmed to operate under the following conditions.

2.1 Hardware Requirements

This FIT module requires your MCU supports the following feature:

- RIICHS

2.2 Software Requirements

This FIT module is dependent upon the following FIT modules:

- Board Support Package Module (r_bsp) Rev.6.10 or higher

2.3 Supported Toolchains

This FIT module is tested and works with the following toolchain:

- Renesas RX Toolchain v.3.03.00
- Renesas RX Toolchain v.3.04.00
- Renesas RX Toolchain v.3.06.00
- Renesas RX Toolchain v.3.07.00

Refer to 6.4 Operating Test Environment for details.

2.4 Usage of Interrupt Vector

The EEI interrupt, RXI interrupt, TXI interrupt, and TEI interrupt are enabled by execution of R_RIICHS_MasterSend function, R_RIICHS_MasterReceive function, or R_RIICHS_SlaveTransfer function (with specified condition)(while the macro definition RIICHS_CFG_CH0_INCLUDE is 1).

Table 2.1 lists the interrupt vector used in the RIICHS FIT Module.

Table 2.1 Interrupt Vector used in the RIICHS FIT Module

Device	Contents
RX671	RXI0 interrupt [channel 0] (vector no.: 118) TXI0 interrupt [channel 0] (vector no.: 119) GROUPAL1 interrupt (vector no.: 113) • TEI0 interrupt [channel 0] (group interrupt source no.: 12) • EEI0 interrupt [channel 0] (group interrupt source no.: 13)

2.5 Header Files

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

2.6 Integer Types

This project uses ANSI C99. These types are defined in stdint.h.

2.7 Configuration Overview

The configuration options in this module are specified in r_riichs_rx_config.h and r_riichs_rx_pin_config.h. The option names and setting values are listed in the table below.

Configuration options in *r_riichs_rx_config.h*	
RIICHS_CFG_PARAM_CHECKING_ENABLE **- Default value = 1**	Selects whether to include parameter checking in the code. - When this is set to 0, parameter checking is omitted. With this setting, the code size can be reduced. - When this is set to 1, parameter checking is included.
RIICHS_CFG_CH0_INCLUDED (1) **- Default value = 1**	Selects whether to use available channels. When not using the channel, set this to 0. - When this is set to 0, relevant processes for the channel are omitted from the code. - When this is set to 1, relevant processes for the channel are included in the code.
RIICHS_CFG_PORT_SET_PROCESSING **- Default value = 1**	Specifies whether to include processing for port setting (*) in the code. * Processing for port setting is the setting to use ports selected by R_RIICHS_CFG_RIICHS0_SCL0_PORT, R_RIICHS_CFG_RIICHS0_SCL0_BIT, R_RIICHS_CFG_RIICHS0_SDA0_PORT, and R_RIICHS_CFG_RIICHS0_SDA0_BIT as pins SCL and SDA. - When this is set to 0, processing for port setting is omitted from the code. - When this is set to 1, processing for port setting is included in the code.

Note:
This setting is invalid for target devices that do not support the corresponding channel.

Configuration options in *r_riichs_rx_pin_config.h*	
R_RIICHS_CFG_RIICHS0_SCL0_PORT **- Default value = '1'**	Selects port groups used as the SCL pins. Specify the value as an ASCII code in the range '0' to 'J'.
R_RIICHS_CFG_RIICHS0_SCL0_BIT **- Default value = '2'**	Selects pins used as the SCL pins. Specify the value as an ASCII code in the range '0' to '7'.
R_RIICHS_CFG_RIICHS0_SDA0_PORT **- Default value = '1'**	Selects port groups used as the SDA pins. Specify the value as an ASCII code in the range '0' to 'J'.
R_RIICHS_CFG_RIICHS0_SDA0_BIT **- Default value = '3'**	Selects pins used as the SDA pins. Specify the value as an ASCII code in the range '0' to '7'.

2.8 Code Size

The ROM (code and constants) and RAM (global data) sizes are determined by the build-time configuration options described in 2.7, Configuration Overview. The table lists reference values when compile options of the C compiler (described in 2.3, Supported Toolchains) are set to their default values. The compile option default values are optimization level: 2, optimization type: for size, and data endianness: little-endian. The code size varies depending on the C compiler version and compile options.

The values in the table below are confirmed under the following conditions.

Module Revision: r_riichs_rx rev1.00

Compiler Version: Renesas Electronics C/C++ Compiler Package for RX Family V3.03.00

(The option of "-lang = c99" is added to the default settings of the integrated development environment.)

GCC for Renesas RX 8.03.00.202002

(The option of "-std=gnu99" is added to the default settings of the integrated development environment.)

IAR C/C++ Compiler for Renesas RX version 4.14.1

(The default settings of the integrated development environment.)

Configuration Options: Default settings

ROM, RAM and Stack Code Sizes

Device	Category	Memory Used					
		Renesas Compiler		GCC		IAR Compiler	
		With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking	With Parameter Checking	Without Parameter Checking
RX671	ROM	1,0411 bytes	9,936 bytes	1,9572 bytes	1,8804 bytes	1,4352 bytes	1,3698 bytes
	RAM	62 bytes		64 bytes		51 bytes	
	STACK*1	344 bytes		-		356 bytes	

Note1. The sizes of maximum usage stack of Interrupts functions is included.

2.9 Parameters

This section describes the structure whose members are API parameters. This structure is located in r_riichs_rx_if.h as are the prototype declarations of API functions.

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIICHS_COMMUNICATION).

```
typedef volatile struct
{
   uint8_t rsv2; /* Reserved area */
   uint8_t rsv1; /* Reserved area */
   riichs_ch_dev_status_t dev_sts; /* Device state flag */
   uint8_t ch_no; /* Channel number of the used device */
   riichs_callback callbackfunc; /* Callback function */
   uint32_t cnt2nd; /* Second data counter (number of bytes) */
   uint32_t cnt1st; /* First data counter (number of bytes) */
   uint32_t *p_data2nd; /* Pointer to the second data storage buffer */
   uint32_t *p_data1st; /* Pointer to the first data storage buffer */
   uint32_t *p_slv_adr; /* Pointer to the slave address storage buffer */
   double scl_up_time; /* Rise time of SCLn Line */
   double scl_down_time; /* Fall time of SCLn Line */
   double fs_scl_up_time; /* Rise time of SCLn Line before transition to Hs mode */
   double fs_scl_down_time; /* Fall time of SCLn Line before transition to Hs mode */
   uint32_t speed_kbps; /* RIICHS bps(kbps) */
   uint32_t fs_speed_kbps; /* RIICHS bps(kbps) before transition to Hs mode */
   uint32_t bus_check_counter; /* software bus busy check counter */
   uint32_t bus_free_time; /* software bus free counter */
   uint16_t slave_addr0; /* Slave Address 0 */
   uint16_t slave_addr1; /* Slave Address 1 */
   uint16_t slave_addr2; /* Slave Address 2 */
   riichs_addr_format_t slave_addr0_format; /* slave address 0 format */
   riichs_addr_format_t slave_addr1_format; /* slave address 1 format */
   riichs_addr_format_t slave_addr2_format; /* slave address 2 format */
   riichs_gca_t gca_enable; /* General call address detection */
   riichs_priority_t rxi_priority; /* The priority level of the RXI */
   riichs_priority_t txi_priority; /* The priority level of the TXI */
   riichs_priority_t eei_priority; /* The priority level of the EEI, not lower than the
   level of TXI and RXI */
   riichs_priority_t tei_priority; /* The priority level of the TEI, not lower than the
   level of TXI and RXI */
   riichs_master_arb_t master_arb; /* Master Arbitration-Lost Detection function */
   riichs_filter_t filter_stage; /* using digital noise filter stage */
   riichs_timeout_t timeout_enable; /* Timeout function */
   riichs_nack_detc_t nack_detc_enable; /* NACK Detection */
   riichs_arb_lost_t arb_lost_enable; /* Arbitration Lost */
   riichs_counter_bit_t counter_bit; /* bit for the timeout detection time */
   riichs_low_count_t l_count; /* SCL line is held LOW when the timeout function is
   enabled */
   riichs_high_count_t h_count; /* SCL line is held HIGH when the timeout function is
   enabled */
   riichs_time_mode_t timeout_mode; /* Timeout Detection Mode Select */
} riichs_info_t;
```

2.10 Return Values

This section describes return values of API functions. This enumeration is located in r_riichs_rx_if.h as are the prototype declarations of API functions.

```
typedef enum
{
   RIICHS_SUCCESS = 0U, /* Function processing completed successfully */
   RIICHS_ERR_LOCK_FUNC, /* The RIICHS is used by another module */
   RIICHS_ERR_INVALID_CHAN, /* Nonexistent channel is specified */
   RIICHS_ERR_INVALID_ARG, /* Invalid parameter is specified */
   RIICHS_ERR_NO_INIT, /* Uninitialized state */
   RIICHS_ERR_BUS_BUSY, /* Bus is busy */
   RIICHS_ERR_AL, /* Arbitration lost error */
   RIICHS_ERR_TMO, /* Timeout is detected */
   RIICHS_ERR_OTHER, /* Other error */
} riichs_return_t;
```

2.11 Callback Functions

In this module, a callback function set up by the user is called when either of the following conditions is met and an EEI interrupt request occurs.

(1) The communication operation (Master Transmission, Master Reception, Master Transmit/Receive, Slave Transmission, Slave Reception) is completed and stop condition is issued.

(2) A timeout was detected during communication operation (Master Transmission, Master Reception, Master Transmit/Receive, Slave Transmission, Slave Reception). (1)

Note:
1. The timeout detection function is enabled in paramter in section 2.9, Parameters.

The callback function is set up by storing the address of the callback function in the callbackfunc structure member described in section 2.9, Parameters and then calling function R_RIICHS_MasterSend(), R_RIICHS_MasterReceive(), R_RIICHS_SlaveTransfer().

API function calls except for the R_RIICHS_GetStatus function is prohibited within a callback function.

2.12 Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends the method using the Smart Configurator described in (1) or (3) or (5) below. However, the Smart Configurator only supports some RX devices. Please use the methods of (2) or (4) for RX devices that are not supported by the Smart Configurator.

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

(2) Adding the FIT module to your project using the FIT Configurator in e2 studio
By using the FIT Configurator in e2 studio, the FIT module is automatically added to your project. Refer to “RX Family Adding Firmware Integration Technology Modules to Projects (R01AN1723)” for details.

(3) Adding the FIT module to your project using the Smart Configurator in CS+
By using the Smart Configurator Standalone version in CS+, the FIT module is automatically added to your project. Refer to “RX Smart Configurator User’s Guide: CS+ (R20AN0470)” for details.

(4) Adding the FIT module to your project in CS+
In CS+, please manually add the FIT module to your project. Refer to “RX Family Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)” for details.

(5) Adding the FIT module to your project using the Smart Configurator in IAREW
By using the Smart Configurator Standalone version, the FIT module is automatically added to your project. Refer to “RX Smart Configurator User’s Guide: IAREW (R20AN0535)” for details.

2.13 “for”, “while” and “do while” statements

In this module, “for”, “while” and “do while” statements (loop processing) are used in processing to wait for register to be reflected and so on. For these loop processing, comments with “WAIT_LOOP” as a keyword are described. Therefore, if user incorporates fail-safe processing into loop processing, user can search the corresponding processing with “WAIT_LOOP”.

The following shows example of description.

while statement example :

```
/* WAIT_LOOP */
while(0 == SYSTEM.OSCOVFSR.BIT.PLOVF)
{
    /* The delay period needed is to make sure that the PLL has stabilized. */
}
```

for statement example :

```
/* Initialize reference counters to 0. */
/* WAIT_LOOP */
for (i = 0; i < BSP_REG_PROTECT_TOTAL_ITEMS; i++)
{
    g_protect_counters[i] = 0;
}
```

do while statement example :

```
/* Reset completion waiting */
do
{
    reg = phy_read(ether_channel, PHY_REG_CONTROL);
    count++;
} while ((reg & PHY_CONTROL_RESET) && (count < ETHER_CFG_PHY_DELAY_RESET)); /* WAIT_LOOP */
```

3. API Functions

R_RIICHS_Open()

This function initializes the RIICHS FIT module. This function must be called before calling any other API functions.

Format

riichs_ return_t R_RIICHS_Open(
    riichs_info_t * p_riichs_info /* Structure data */
)

Parameters

**p_riichs_info*

This is the pointer to the I²C communication information structure.
Only the member of the structure used in this function is described here. Refer to 2.9 Parameters for details on the structure.

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIICHS_COMMUNICATION) and when an error has occurred (RIICHS_TMO and RIICHS_ERROR).

For the parameter which has '(to be updated)' in the comment below, the argument for the parameter will be updated during the API execution.

```
riichs_ch_dev_status_t dev_sts; /* Device state flag (to be updated) */
uint8_t ch_no;  /* Channel number */
double scl_up_time; /* Rise time of SCLn Line */
double scl_down_time; /* Fall time of SCLn Line */
double fs_scl_up_time; /* Rise time of SCLn Line before transition to Hs mode */
double fs_scl_down_time; /* Fall time of SCLn Line before transition to Hs mode */
uint32_t speed_kbps; /* RIICHS bps(kbps) */
uint32_t fs_speed_kbps; /* RIICHS bps(kbps) before transition to Hs mode */
uint32_t bus_check_counter; /* software bus busy check counter */
uint32_t bus_free_time; /* software bus free counter */
uint16_t slave_addr0; /* Slave Address 0 */
uint16_t slave_addr1; /* Slave Address 1 */
uint16_t slave_addr2; /* Slave Address 2 */
riichs_addr_format_t slave_addr0_format; /* slave address 0 format */
riichs_addr_format_t slave_addr1_format; /* slave address 1 format */
riichs_addr_format_t slave_addr2_format; /* slave address 2 format */
riichs_gca_t gca_enable; /* General call address detection */
riichs_priority_t rxi_priority; /* The priority level of the RXI */
riichs_priority_t txi_priority; /* The priority level of the TXI */
riichs_priority_t eei_priority; /* The priority level of the EEI,
                                   not lower than the level of TXI and RXI */
riichs_priority_t tei_priority; /* The priority level of the TEI,
                                   not lower than the level of TXI and RXI */
riichs_master_arb_t master_arb; /* Master Arbitration-Lost Detection function */
riichs_filter_t filter_stage; /* using digital noise filter stage */
riichs_timeout_t timeout_enable; /* Timeout function */
riichs_nack_detc_t nack_detc_enable; /* NACK Detection */
riichs_arb_lost_t arb_lost_enable; /* Arbitration Lost */
riichs_counter_bit_t counter_bit; /* bit for the timeout detection time */
riichs_low_count_t l_count; /* SCL line is held LOW when the timeout function is
                                   enabled */
```

```
    riichs_high_count_t h_count; /* SCL line is held HIGH when the timeout function is
                                    enabled */
    riichs_time_mode_t timeout_mode; /* Timeout Detection Mode Select */
```

Return Values

RIICHS_SUCCESS */* Processing completed successfully */*
RIICHS_ERR_LOCK_FUNC */* The API is locked by the other task. */*
RIICHS_ERR_INVALID_CHAN */* Nonexistent channel */*
RIICHS_ERR_INVALID_ARG */* Invalid parameter */*
RIICHS_ERR_OTHER */* The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riichs_rx_if.h.

Description

Performs the initialization to start the RIICHS communication. Sets the RIICHS channel specified by the parameter. If the state of the channel is ‘uninitialized (RIICHS_NO_INIT)’, the following processes are performed.

- Setting the state flag
- Setting I/O ports
- Allocating I²C output ports
- Cancelling RIICHS module-stop state
- Initializing variables used by the API
- Initializing the RIICHS registers used for the RIICHS communication
- Disabling the RIICHS interrupts

Example

```
volatile riichs_return_t  ret;
riichs_info_t             iichs_info_m;

iichs_info_m.dev_sts = RIICHS_NO_INIT;
iichs_info_m.ch_no   = 0;
iichs_info_m.scl_up_time = 20E-9;
iichs_info_m.scl_down_time = 20E-9;
iichs_info_m.fs_scl_up_time = 20E-9;
iichs_info_m.fs_scl_down_time = 20E-9;
iichs_info_m.speed_kbps = 3400;
iichs_info_m.fs_speed_kbps = 400;
iichs_info_m.bus_check_counter = 1000;
iichs_info_m.bus_free_time = 5;
iichs_info_m.slave_addr0 = 0x0025;
iichs_info_m.slave_addr1 = 0x0000;
iichs_info_m.slave_addr2 = 0x0000;
iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_m.gca_enable = RIICHS_GCA_DISABLE;
iichs_info_m.rxi_priority = RIICHS_IPL_1;
iichs_info_m.txi_priority = RIICHS_IPL_1;
iichs_info_m.eei_priority = RIICHS_IPL_1;
iichs_info_m.tei_priority = RIICHS_IPL_1;
iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0;
iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE;
iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16;
iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE;
iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE;
iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;

ret = R_RIICHS_Open(&iichs_info_m);
```

Special Notes
The table below lists available settings.

Structure Member	Available Settings for Each Pattern of the Master Transmission
scl_up_time	Rise time of SCLn Line.
scl_down_time	Fall time of SCLn Line.
fs_scl_up_time	Rise time of SCLn Line before transition to Hs mode.
fs_scl_down_time	Fall time of SCLn Line before transition to Hs mode.
speed_kbps	1 to 3400
fs_speed_kbps	0 to 1000 (1)
bus_check_counter	0000 0000h to 0000 ffffh
bus_free_time	0000 0000h to 0000 01ffh
slave_addr0	This set the slave address 0. (2)
slave_addr1	This set the slave address 1. (2)
slave_addr2	This set the slave address 2. (2)
slave_addr0_format	RIICHS_ADDR_FORMAT_NONE (slave address is not set), RIICHS_SEVEN_BIT_ADDR_FORMAT (7-bit slave address format is set.) or RIICHS_TEN_BIT_ADDR_FORMAT (10-bit slave address format is set.)
slave_addr1_format	RIICHS_ADDR_FORMAT_NONE (slave address is not set), RIICHS_SEVEN_BIT_ADDR_FORMAT (7-bit slave address format is set.) or RIICHS_TEN_BIT_ADDR_FORMAT (10-bit slave address format is set.)
slave_addr2_format	RIICHS_ADDR_FORMAT_NONE (slave address is not set), RIICHS_SEVEN_BIT_ADDR_FORMAT (7-bit slave address format is set.) or RIICHS_TEN_BIT_ADDR_FORMAT (10-bit slave address format is set.)
gca_enable	RIICHS_GCA_ENABLE or RIICHS_GCA_DISABLE
rxi_priority	RIICHS_IPL_1 to RIICHS_IPL_15
txi_priority	RIICHS_IPL_1 to RIICHS_IPL_15
eei_priority	RIICHS_IPL_1 to RIICHS_IPL_15 (3)
tei_priority	RIICHS_IPL_1 to RIICHS_IPL_15 (3)
master_arb	RIICHS_MASTER_ARB_LOST_ENABLE or RIICHS_MASTER_ARB_LOST_DISABLE
filter_stage	RIICHS_DIGITAL_FILTER_0 to RIICHS_DIGITAL_FILTER_16
timeout_enable	RIICHS_TMO_ENABLE or RIICHS_TMO_DISABLE
nack_detc_enable	RIICHS_NACK_DETC_ENABLE or RIICHS_NACK_DETC_DISABLE
arb_lost_enable	RIICHS_ARB_LOST_ENABLE or RIICHS_ARB_LOST_DISABLE
counter_bit	RIICHS_COUNTER_BIT6, RIICHS_COUNTER_BIT8, RIICHS_COUNTER_BIT14 or RIICHS_COUNTER_BIT16
l_count	RIICHS_L_COUNT_ENABLE or RIICHS_L_COUNT_DISABLE
h_count	RIICHS_H_COUNT_ENABLE or RIICHS_H_COUNT_DISABLE
timeout_mode	RIICHS_TIMEOUT_MODE_ALL, RIICHS_TIMEOUT_MODE_BUSY or RIICHS_TIMEOUT_MODE_FREE

Notes:
1. If speed_kbps exceeds 1000, 0 cannot be set.
2. Available bits of the setting value vary depending on the setting value of the slave_addrj_format. (j = 0 to 2)
3. Do not set this option to a value lower than the priority level specified with RIICHS_CFG_CH0_RXI_INT_PRIORITY or RIICHS_CFG_CH0_TXI_INT_PRIORITY.

R_RIICHS_MasterSend()

Starts master transmission. Changes the transmit pattern according to the parameters. Operates batched processing until stop condition generation.

Format

riichs_return_t R_RIICHS_MasterSend(

riichs_info_t * p_riichs_info /* Structure data */

)

Parameters

**p_riichs_info*

This is the pointer to the I²C communication information structure. The transmit patterns can be selected from four patterns by the parameter setting. Refer to Special Notes in this section for available settings and the setting values for each transmit pattern. Also refer to 1.3.2 Master Transmission for details of each pattern.
Only members of the structure used in this function are described here. Refer to 2.9 Parameters for details on the structure.

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIICHS_COMMUNICATION) and when an error has occurred (RIICHS_TMO and RIICHS_ERROR).

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

For the parameter which has '(to be updated)' in the comment below, the argument for the parameter will be updated during the API execution.

```
riichs_ch_dev_status_t dev_sts; /* Device state flag (to be updated)*/
uint8_t ch_no; /* Channel number */
riichs_callback callbackfunc; /* Callback function */
uint32_t cnt2nd; /* Second data counter (number of bytes)
                                    (to be updated for only pattern 1 and 2) */
uint32_t cnt1st; /* First data counter (number of bytes)
                                    (to be updated for only pattern 1) */
uint32_t * p_data2nd; /* Pointer to the second data storage buffer */
uint32_t * p_data1st; /* Pointer to the first data storage buffer */
uint32_t * p_slv_adr; /* Pointer to the slave address storage buffer */
```

Return Values

RIICHS_SUCCESS	*/* Processing completed successfully */*
RIICHS_ERR_INVALID_CHAN	*/* The channel is nonexistent. */*
RIICHS_ERR_INVALID_ARG	*/* The parameter is invalid. */*
RIICHS_ERR_NO_INIT	*/* Uninitialized state */*
RIICHS_ERR_BUS_BUSY	*/* The bus state is busy. */*
RIICHS_ERR_AL	*/* Arbitration-lost error occurred */*
RIICHS_ERR_TMO	*/* Timeout is detected */*
RIICHS_ERR_OTHER	*/* The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riichs_rx_if.h.

Description

Starts the RIICHS master transmission. The transmission is performed with the RIICHS channel and transmit pattern specified by parameters. If the state of the channel is ‘idle (RIICHS_IDLE, RIICHS_FINISH, or RIICHS_NACK)’, the following processes are performed.

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

This function returns RIICHS_SUCCESS as a return value when the processing up to the start condition generation ends normally. This function returns RIICHS_ERR_BUS_BUSY as a return value when the following conditions are met to the start condition generation ends normally. (1)

- The internal status bit is in busy state.
- Either SCL or SDA line is in low state.

The transmission processing is performed sequentially in subsequent interrupt processing after this function return RIICHS_SUCCESS. Section "2.4 Usage of Interrupt Vector" should be refered for the interrupt to be used. For master transmission, the interrupt generation timing should be refered from "6.2.1 Master transmission".

After issuing a stop condition at the end of transmission, the callback function specified by the argument is called.

The transmission completion is performed normally or not, can be confirmed by checking the device status flag specified by the argument or the channel status flag g_riichs_ChStatus [], that is to be "RIICHS_FINISH" for normal completion.

Notes:

1. When SCL and SDA pin is not external pull-up, this function may return RIICHS_ERR_BUS_BUSY by detecting either SCL or SDA line is as in low state.

Example

```
/* for MasterSend(Pattern 1) */
#include <stddef.h>
#include "platform.h"
#include "r_riichs_rx_if.h"

riichs_info_t iichs_info_m;

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

void main(void)
{
    volatile riichs_return_t ret;

    uint8_t addr_eeprom[1]   = {0x53};
    uint8_t access_addr1[1]  = {0x00};
    uint8_t mst_send_data[5] = {0x81,0x82,0x83,0x84,0x85};

    /* Sets IICHS Information for sending pattern 1. */
    iichs_info_m.dev_sts = RIICHS_NO_INIT;
    iichs_info_m.ch_no = 0;
    iichs_info_m.callbackfunc = &CallbackMaster;
```

```
    iichs_info_m.cnt2nd = 3;
    iichs_info_m.cnt1st = 1;
    iichs_info_m.p_data2nd = mst_send_data;
    iichs_info_m.p_data1st = access_addr1;
    iichs_info_m.p_slv_adr = addr_eeprom;
    iichs_info_m.scl_up_time = 20E-9;
    iichs_info_m.scl_down_time = 20E-9;
    iichs_info_m.fs_scl_up_time = 20E-9;
    iichs_info_m.fs_scl_down_time = 20E-9;
    iichs_info_m.speed_kbps = 3400;
    iichs_info_m.fs_speed_kbps = 400;
    iichs_info_m.bus_check_counter = 1000;
    iichs_info_m.bus_free_time = 5;
    iichs_info_m.slave_addr0 = 0x0025;
    iichs_info_m.slave_addr1 = 0x0000;
    iichs_info_m.slave_addr2 = 0x0000;
    iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
    iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_m.gca_enable = RIICHS_GCA_DISABLE;
    iichs_info_m.rxi_priority = RIICHS_IPL_1;
    iichs_info_m.txi_priority = RIICHS_IPL_1;
    iichs_info_m.eei_priority = RIICHS_IPL_1;
    iichs_info_m.tei_priority = RIICHS_IPL_1;
    iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
    iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0;
    iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE;
    iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
    iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
    iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16;
    iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE;
    iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE;
    iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;

    /* RIICHS open */
    ret = R_RIICHS_Open(&iichs_info_m);

    /* RIICHS send start */
    ret = R_RIICHS_MasterSend(&iichs_info_m);

    if (RIICHS_SUCCESS == ret)
    {
        while(RIICHS_FINISH != iichs_info_m.dev_sts);
    }
    else
    {
        /* error */
    }

    /* RIICHS send complete */
    while(1);
}

void CallbackMaster(void)
{
    volatile riichs_return_t ret;
    riichs_mcu_status_t      iichs_status;
```

```
    ret = R_RIICHS_GetStatus(&iichs_info_m, &iichs_status);
    if(RIICHS_SUCCESS != ret)
    {
        /* Call error processing for the R_RIICHS_GetStatus() function */
    }
    else
    {
        /* Processing when a timeout, arbitration-lost, NACK,
           or others is detected by verifying the iichs_status flag. */
    }
}
```

Special Notes

The table below lists available settings for each pattern.

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

Notes:

1. When using pattern 2, 3, set ‘FIT_NO_PTR’ as the argument of the parameter.
2. 0 cannot be set.

R_RIICHS_MasterReceive()

Starts master reception. Changes the receive pattern according to the parameters. Operates batched processing until stop condition generation.

Format

riichs_return_t R_RIICHS_MasterRecive(riichs_info_t * p_riichs_info /* Structure data */)

Parameters

**p_riichs_info*

This is the pointer to the I²C communication information structure. The receive pattern can be selected from master reception and master transmit/receive by the parameter setting. Refer to the Special Notes in this section for available settings and the setting values for each receive pattern. Also refer to 1.3.3 Master Reception for details of each receive pattern.
Only members of the structure used in this function are described here. Refer to 2.9 Parameters for details on the structure.

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIICHS_COMMUNICATION) and when an error has occurred (RIICHS_TMO and RIICHS_ERROR).

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

For the parameter which has '(to be updated)' in the comment below, the argument for the parameter will be updated during the API execution.

```
riichs_ch_dev_status_t dev_sts; /* Device state flag (to be updated) */
uint8_t ch_no; /* Channel number */
riichs_callback callbackfunc; /* Callback function */
uint32_t cnt2nd;/* Second data counter (number of bytes) (to be updated) */
uint32_t cnt1st;/* First data counter (number of bytes)
                            (to be updated only for master transmit/receive) */
uint32_t * p_data2nd; /* Pointer to the second data storage buffer */
uint32_t * p_data1st; /* Pointer to the first data storage buffer */
uint32_t * p_slv_adr; /* Pointer to the slave address storage buffer */
```

Return Values

RIICHS_SUCCESS	*/* Processing completed successfully */*
RIICHS_ERR_INVALID_CHAN	*/* The channel is nonexistent. */*
RIICHS_ERR_INVALID_ARG	*/* The parameter is invalid. */*
RIICHS_ERR_NO_INIT	*/* Uninitialized state */*
RIICHS_ERR_BUS_BUSY	*/* The bus state is busy. */*
RIICHS_ERR_AL	*/* Arbitration-lost error occurred */*
RIICHS_ERR_TMO	*/* Timeout is detected */*
RIICHS_ERR_OTHER	*/* The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riichs_rx_if.h.

Description

Starts the RIICHS master reception. The reception is performed with the RIICHS channel and receive pattern specified by parameters. If the state of the channel is 'idle (RIICHS_IDLE, RIICHS_FINISH, or RIICHS_NACK)', the following processes are performed.

- Setting the state flag

- Initializing variables used by the API
- Enabling the RIICHS interrupts
- Generating a start condition

This function returns RIICHS_SUCCESS as a return value when the processing up to the start condition generation ends normally. This function returns RIICHS_ERR_BUS_BUSY as a return value when the following conditions are met to the start condition generation ends normally. (1)

- The internal status bit is in busy state.
- Either SCL or SDA line is in low state.

The reception processing is performed sequentially in subsequent interrupt processing after this function return RIICHS_SUCCESS. Section "2.4 Usage of Interrupt Vector" should be refered for the interrupt to be used. For master transmission, the interrupt generation timing should be refered from "6.2.2 Master Reception".

After issuing a stop condition at the end of reception, the callback function specified by the argument is called.

The reception completion is performed normally or not, can be confirmed by checking the device status flag specified by the argument or the channel status flag g_riichs_ChStatus [], that is to be "RIICHS_FINISH" for normal completion.

Notes:

1. When SCL and SDA pin is not external pull-up, this function may return RIICHS_ERR_BUS_BUSY by detecting either SCL or SDA line is as in low state.

Example

```
#include <stddef.h>
#include "platform.h"
#include "r_riichs_rx_if.h"

riichs_info_t    iichs_info_m;

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

void main(void)
{
    volatile riichs_return_t ret;

    uint8_t addr_eeprom[1]    = {0x53};
    uint8_t access_addr1[1]   = {0x00};
    uint8_t mst_store_area[5] = {0xFF,0xFF,0xFF,0xFF,0xFF};

    /* Sets IICHS Information. */
    iichs_info_m.dev_sts = RIICHS_NO_INIT;
    iichs_info_m.ch_no = 0;
    iichs_info_m.callbackfunc = &CallbackMaster;
    iichs_info_m.cnt2nd = 3;
    iichs_info_m.cnt1st = 1;
    iichs_info_m.p_data2nd = mst_store_area;
    iichs_info_m.p_data1st = access_addr1;
    iichs_info_m.p_slv_adr = addr_eeprom;
    iichs_info_m.scl_up_time = 20E-9;
    iichs_info_m.scl_down_time = 20E-9;
```

```
    iichs_info_m.fs_scl_up_time = 20E-9;
    iichs_info_m.fs_scl_down_time = 20E-9;
    iichs_info_m.speed_kbps = 3400;
    iichs_info_m.fs_speed_kbps = 400;
    iichs_info_m.bus_check_counter = 1000;
    iichs_info_m.bus_free_time = 5;
    iichs_info_m.slave_addr0 = 0x0025;
    iichs_info_m.slave_addr1 = 0x0000;
    iichs_info_m.slave_addr2 = 0x0000;
    iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
    iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_m.gca_enable = RIICHS_GCA_DISABLE;
    iichs_info_m.rxi_priority = RIICHS_IPL_1;
    iichs_info_m.txi_priority = RIICHS_IPL_1;
    iichs_info_m.eei_priority = RIICHS_IPL_1;
    iichs_info_m.tei_priority = RIICHS_IPL_1;
    iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
    iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0;
    iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE;
    iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
    iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
    iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16;
    iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE;
    iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE;
    iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;

    /* RIICHS open */
    ret = R_RIICHS_Open(&iichs_info_m);

    /* RIICHS receive start */
    ret = R_RIICHS_MasterReceive(&iichs_info_m);

    if (RIICHS_SUCCESS == ret)
    {
        while(RIICHS_FINISH != iichs_info_m.dev_sts);
    }
    else
    {
        /* error */
    }
    /* RIICHS receive complete */
    while(1);
}

void CallbackMaster(void)
{
    volatile riichs_return_t ret;
    riichs_mcu_status_t      iichs_status;

    ret = R_RIICHS_GetStatus(&iichs_info_m, &iichs_status);
    if(RIICHS_SUCCESS != ret)
    {
        /* Call error processing for the R_RIICHS_GetStatus() function */
    }
    else
    {
```

```
        /* Processing when a timeout, arbitration-lost, NACK,
           or others is detected by verifying the iichs_status flag._*/
    }
}
```

Special Notes

The table below lists available settings for each receive pattern.

Structure Member	Available Settings for Each Pattern of the Master Reception	
	Master Reception	Master transmit/receive
*p_slv_adr	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 transmitting
*p_data2nd	Pointer to the second data storage buffer for receiving	
dev_sts	Device state flag	
cnt1st (1)	0	0000 0001h to FFFF FFFFh
cnt2nd	0000 0001h to FFFF FFFFh (2)	
callbackfunc	Specify the function name used	
ch_no	00h to FFh	
rsv1, rsv2	Reserved (value set here has no effect)	

Notes:
1. The receive pattern is determined by whether cnt1st is 0 or not.
2. 0 cannot be set.

R_RIICHS_SlaveTransfer()

This function performs slave transmission and reception. Changes the transmit and receive pattern according to the parameters.

Format

riichs_return_t R_RIICHS_SlaveTransfer(

riichs_info_t * p_riichs_info /* Structure data */

)

Parameters

**p_riichs_info*

This is the pointer to the I²C communication information structure. The operation can be selected from preparation for slave reception, slave transmission, or both of them by the parameter setting. Refer to the Special Notes in this section for available parameter settings. Also refer to 1.3.4 Slave Transmission and Reception for details of slave operations.
Only members of the structure used in this function are described here. Refer to 2.9 Parameters for details on the structure.

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIICHS_COMMUNICATION) and when an error has occurred (RIICHS_TMO and RIICHS_ERROR).

For the parameter which has '(to be updated)' in the comment below, the argument for the parameter will be updated during the API execution.

```
riichs_ch_dev_status_t dev_sts; /* Device state flag (to be updated)*/
uint8_t ch_no; /* Channel number */
riichs_callback callbackfunc; /* Callback function */
uint32_t cnt2nd; /* Second data counter (number of bytes)
                            (to be updated for only slave reception) */
uint32_t cnt1st; /* First data counter (number of bytes)
                            (to be updated for only slave transmission) */
uint32_t * p_data2nd; /* Pointer to the second data storage buffer */
uint32_t * p_data1st; /* Pointer to the first data storage buffer */
```

Return Values

RIICHS_SUCCESS	*/* Processing completed successfully */*
RIICHS_ERR_INVALID_CHAN	*/* The channel is nonexistent. */*
RIICHS_ERR_INVALID_ARG	*/* The parameter is invalid. */*
RIICHS_ERR_NO_INIT	*/* Uninitialized state */*
RIICHS_ERR_BUS_BUSY	*/* The bus state is busy. */*
RIICHS_ERR_AL	*/* Arbitration-lost error occurred */*
RIICHS_ERR_TMO	*/* Timeout is detected */*
RIICHS_ERR_OTHER	*/* The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riichs_rx_if.h.

Description

Prepares for the RIICHS slave transmission or slave reception. If this function is called while the master is communicating, an error occurs. Sets the RIICHS channel specified by the parameter. If the state of the channel is 'idle (RIICHS_IDLE, RIICHS_FINISH, or RIICHS_NACK)', the following processes are performed.

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

This function returns RIICHS_SUCCESS as a return value when the setting of slave address and permission of slave address match interrupt are completed normally.

The processing of slave transmission or slave reception is performed sequentially in the subsequent interrupt processing.

Section "2.4 Usage of Interrupt Vector" should be refered for the interrupt to be used.

The interrupt generation timing of slave transmission should be refered from "6.2.4 Slave Transmission". The interrupt generation timing for slave reception should be refered from "6.2.5 Slave reception".

After detecting the stop condition of slave transmission or slave reception termination, the callback function specified by the argument is called.

The successful completion of slave reception can be checked by confirming the device status flag or channel status flag specified in the argument g_riichs_ChStatus [], that is to be "RIICHS_FINISH". The successful completion of slave transmission can be checked by confirming the device status flag or channel status flag specified in the argument g_riichs_ChStatus [], that is to be "RIICHS_FINISH" or "RIICHS_NACK". "RIICHS_NACK" is set when master device transmitted NACK for notify to the slave that last data receive completed.

Example

```
#include <stddef.h>
#include "platform.h"
#include "r_riichs_rx_if.h"

riichs_info_t    iichs_info_m;

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

void main(void)
{
    volatile    riichs_return_t ret;
    riichs_info_t iichs_info_s;

    uint8_t addr_eeprom[1]    = {0x25};
    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 IICHS Information for Master Send. */
```

```
    iichs_info_m.dev_sts = RIICHS_NO_INIT;
    iichs_info_m.ch_no = 0;
    iichs_info_m.callbackfunc = &CallbackMaster;
    iichs_info_m.cnt2nd = 3;
    iichs_info_m.cnt1st = 1;
    iichs_info_m.p_data2nd = mst_store_area;
    iichs_info_m.p_data1st = access_addr1;
    iichs_info_m.p_slv_adr = addr_eeprom;
    iichs_info_m.scl_up_time = 20E-9;
    iichs_info_m.scl_down_time = 20E-9;
    iichs_info_m.fs_scl_up_time = 20E-9;
    iichs_info_m.fs_scl_down_time = 20E-9;
    iichs_info_m.speed_kbps = 3400;
    iichs_info_m.fs_speed_kbps = 400;
    iichs_info_m.bus_check_counter = 1000;
    iichs_info_m.bus_free_time = 5;
    iichs_info_m.slave_addr0 = 0x0000;
    iichs_info_m.slave_addr1 = 0x0000;
    iichs_info_m.slave_addr2 = 0x0000;
    iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
    iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_m.gca_enable = RIICHS_GCA_DISABLE;
    iichs_info_m.rxi_priority = RIICHS_IPL_1;
    iichs_info_m.txi_priority = RIICHS_IPL_1;
    iichs_info_m.eei_priority = RIICHS_IPL_1;
    iichs_info_m.tei_priority = RIICHS_IPL_1;
    iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
    iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0;
    iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE;
    iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
    iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
    iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16;
    iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE;
    iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE;
    iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;

    /* Sets IICHS Information for Slave Transfer. */
    iichs_info_s.dev_sts = RIICHS_NO_INIT;
    iichs_info_s.ch_no = 0;
    iichs_info_s.callbackfunc = &CallbackSlave;
    iichs_info_s.cnt2nd = 4;
    iichs_info_s.cnt1st = 3;
    iichs_info_s.p_data2nd = slv_store_area;
    iichs_info_s.p_data1st = slv_send_data;
    iichs_info_s.p_slv_adr = (uint8_t*)FIT_NO_PTR;
    iichs_info_s.scl_up_time = 20E-9;
    iichs_info_s.scl_down_time = 20E-9;
    iichs_info_s.fs_scl_up_time = 20E-9;
    iichs_info_s.fs_scl_down_time = 20E-9;
    iichs_info_s.speed_kbps = 3400;
    iichs_info_s.fs_speed_kbps = 400;
    iichs_info_s.bus_check_counter = 1000;
    iichs_info_s.bus_free_time = 5;
    iichs_info_s.slave_addr0 = 0x0025;
    iichs_info_s.slave_addr1 = 0x0000;
    iichs_info_s.slave_addr2 = 0x0000;
```

```
    iichs_info_s.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
    iichs_info_s.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_s.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_s.gca_enable = RIICHS_GCA_DISABLE;
    iichs_info_s.rxi_priority = RIICHS_IPL_1;
    iichs_info_s.txi_priority = RIICHS_IPL_1;
    iichs_info_s.eei_priority = RIICHS_IPL_1;
    iichs_info_s.tei_priority = RIICHS_IPL_1;
    iichs_info_s.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
    iichs_info_s.filter_stage = RIICHS_DIGITAL_FILTER_0;
    iichs_info_s.timeout_enable = RIICHS_TMO_ENABLE;
    iichs_info_s.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
    iichs_info_s.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
    iichs_info_s.counter_bit = RIICHS_COUNTER_BIT16;
    iichs_info_s.l_count = RIICHS_L_COUNT_ENABLE;
    iichs_info_s.h_count = RIICHS_H_COUNT_ENABLE;
    iichs_info_s.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;

    /* RIICHS open */
    ret = R_RIICHS_Open(&iichs_info_m);

    /* RIICHS slave transfer enable */
    ret = R_RIICHS_SlaveTransfer(&iichs_info_s);

    /* RIICHS master send start */
    ret = R_RIICHS_MasterSend(&iichs_info_m);

    while(1);
}

void CallbackMaster(void)
{
    volatile riichs_return_t ret;
    riichs_mcu_status_t      iichs_status;

    ret = R_RIICHS_GetStatus(&iichs_info_m, &iichs_status);
    if(RIICHS_SUCCESS != ret)
    {
        /* Call error processing for the R_RIICHS_GetStatus() function */
    }
    else
    {
        /* Processing when a timeout, arbitration-lost, NACK,
           or others is detected by verifying the iichs_status flag._*/
    }
}

void CallbackSlave(void)
{
    /* Processing when an event occurs in slave mode as required. */
}
```

Special Notes

The table below lists available settings for each receive pattern.

Structure Member	Available Parameter Settings	
	Slave Reception	Slave Transmission
*p_slv_adr	Not used (value set here has no effect)	
*p_data1st	(For slave transmission)	Pointer to the first data storage buffer for transmitting (1)
*p_data2nd	Pointer to the second data storage buffer for receiving (2)	(For slave reception)
dev_sts	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 used	
ch_no	00h to FFh	
rsv1, rsv2	Reserved (value set here has no effect)	

Notes:

1. Set this when performing slave transmission.
   When slave transmission is not used in the user system, set FIT_NO_PTR.
2. Set this when performing slave reception.
   When slave reception is not used in the user system, set FIT_NO_PTR.

R_RIICHS_GetStatus()

Returns the state of this module.

Format

```
riichs_sts_flg_t R_RIICHS_GetStatus(
        riichs_info_t *   p_riichs_info             /* Structure data */
        riichs_mcu_status_t *  p_riichs_status   /* RIICHS state */
)
```

Parameters

**p_riichs_info*

This is the pointer to the I²C communication information structure.
Only the member of the structure used in this function is described here. Refer to 2.9 Parameters for details on the structure.

For the parameter which has '(to be updated)' in the comment below, the argument for the parameter will be updated during the API execution.

```
riichs_ch_dev_status_t dev_sts; /* Device state flag
                          (to be updated when the state is "RIICHS_AL")*/
uint8_t ch_no; /* Channel number */
```

**p_riichs_status*

This contains the variable to store the RIICHS state. Use the structure members listed below to specify parameters.

```
typedef union
{
   uint32_t LONG;
   struct
   {
          uint32_t rsv :11;  /* 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;   /* General call 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 */
            uint32_t HSMC:1;    /* Hs mode Master Code Detection flag */
      }BIT;
    } riichs_mcu_status_t;
```

Return Values

RIICHS_SUCCESS */* Processing completed successfully */*
RIICHS_ERR_INVALID_CHAN */* The channel is nonexistent. */*
RIICHS_ERR_INVALID_ARG */* The parameter is invalid. */*

Properties

Prototyped in r_riichs_rx_if.h.

Description

Returns the state of this module.
By reading the register, pin level, variable, or others, obtains the state of the RIICHS channel which specified by the parameter, and returns the obtained state as 32-bit structure.

When this function is called, the RIICHS arbitration-lost flag and NACK flag are cleared to 0. If the device state is “RIICHS_ AL”, the value is updated to “RIICHS_FINISH”.

Example

```
volatile riichs_return_t  ret;
riichs_info_t             iichs_info_m;
riichs_mcu_status_t       riichs_status;

iichs_info_m.ch_no = 0;

ret = R_RIICHS_GetStatus(&iichs_info_m, &riichs_status);
```

Special Notes

The following shows the state flag allocation.

b31 to b21	b20	b19	b18	b17	b16
Reserved	Slave Address Detection			Event detection	Event detection
Reserved	Slave Address 0 Detection	Slave Address 1 Detection	Slave Address 2 Detection	General call detection	Device-ID detection
Rsv	AAS2	AAS1	AAS0	GCA	DID
Undefined	0: Not detected 1: Detected			0: Not detected 1:Detected	0: Not detected 1: Detected

b15	b14	b13	b12	b11	b10	b9	b8
Event detection	Mode flag	Event detection	Event detection	Event detection	Event detection	buffer status	buffer status
Host address detection	Master/slave mode	Timeout detection	Arbitration lost detection	Stop detection	Start detection	Receive buffer	Send buffer
HOA	MST	TMO	AL	SP	ST	RBUF	SBUF
0: Not detected 1: Detected	0: slave 1: master	0: Not detected 1: Detected	0: Not detected 1: Detected	0: Not detected 1: Detected	0: Not detected 1: Detected	0: no data 1: have data	0: have data 1: no data

b7	b6	b5	b4	b3	b2	b1	b0
Pin status		Pin level		Event detection	Mode flag	Bus state	Event detection
SCL pin control	SDA pin control	SCL pin level	SDA pin level	NACK detection	Transmit/ Receive mode	Bus busy/ready	Master code detection
SCLO	SDAO	SCLI	SDAI	NACK	TRS	BSY	HSMC
0: Output low level 1: Output Hi-Z		0: Low level 1: High level		0: Not detected 1: Detected	0: Receive 1: Transmit	0: Idle 1: Busy	0: Not detected 1: Detected

R_RIICHS_Control()

This function outputs conditions, Hi-Z from the SDA, and one-shot of the SCL clock. Also it resets the settings of this module. This function is mainly used when a communication error occurs.

Format

```
riichs_return_t R_RIICHS_Control(
        r_riichs_info_t *           p_riichs_info  /* Structure data */
        uint8_t                     ctrl_ptn       /* Output pattern */
);
```

Parameters

**p_riichs_info*

This is the pointer to the I²C communication information structure.
Only the member of the structure used in this function is described here. Refer to 2.9 Parameters for details on the structure.

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIICHS_COMMUNICATION) and when an error has occurred (RIICHS_TMO and RIICHS_ERROR).

For the parameter which has '(to be updated)' in the comment below, the argument for the parameter will be updated during the API execution.

```
riichs_ch_dev_status_t   dev_sts; /* Device state flag
                            (to be updated when "RIICHS_GEN_RESET" is
                            specified as the output pattern )*/
uint8_t ch_no; /* Channel number */
```

ctrl_ptn

Specifies the output pattern.

The output pattern listed below can be specified simultaneously. When specifying multiple patterns simultaneously, specify them with '|'(OR).

The following output patterns can be specified simultaneously with a combination of two or three of them.

- RIICHS_GEN_START_CON
- RIICHS_GEN_RESTART_CON
- RIICHS_GEN_STOP_CON

The following two can specified simultaneously.

- RIICHS_GEN_SDA_HI_Z
- RIICHS_GEN_SCL_ONESHOT

```
#define RIICHS_GEN_START_CON (uint8_t)(0x01) /* Start condition generation */
#define RIICHS_GEN_STOP_CON (uint8_t)(0x02) /* Stop condition generation */
#define RIICHS_GEN_RESTART_CON (uint8_t)(0x04) /* Restart condition generation */
#define RIICHS_GEN_SDA_HI_Z (uint8_t)(0x08) /* Hi-Z output from the SDA pin */
#define RIICHS_GEN_SCL_ONESHOT (uint8_t)(0x10) /* SCL clock one-shot output */
#define RIICHS_GEN_RESET (uint8_t)(0x20) /* RIICHS module reset */
```

Return Values

RIICHS_SUCCESS	*/* Processing completed successfully */*
RIICHS_ERR_INVALID_CHAN	*/* Nonexistent channel */*
RIICHS_ERR_INVALID_ARG	*/* Invalid parameter */*
RIICHS_ERR_BUS_BUSY	*/* Bus is busy */*
RIICHS_ERR_AL	*/* Arbitration-lost error occurred */*
RIICHS_ERR_OTHER	*/* The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riichs_rx_if.h.

Description

Outputs control signals of the RIICHS. Outputs conditions specified by the argument, Hi-Z from the SDA pin, and one-shot of the SCL clock. Also resets the RIICHS module settings.

Example

```
/* Outputs an extra SCL clock cycle after the SDA pin state is changed to a high-
impedance state. */
volatile riichs_return_t  ret;
riichs_info_t             iichs_info_m;

iichs_info_m.ch_no = 0;

ret = R_RIICHS_Control(&iichs_info_m, RIICHS_GEN_SDA_HI_Z | RIICHS_GEN_SCL_ONESHOT);
```

Special Notes

One-shot output of the SCL clock

In master mode, if the clock signals from the master and slave devices go out of synchronization due to noise or other factors, the slave device may hold the SDA line low (bus hang up). Then the SDA line can be released from being held low by outputting one clock of the SCL at a time.

In this module, one clock of the SCL can be output by setting the output pattern “RIICHS_GEN_SCL_ONESHOT” (one-shot output of the SCL clock) and calling R_RIICHS_Control().

R_RIICHS_Close()

This function completes the RIICHS communication and releases the RIICHS used.

Format

riichs_return_t R_RIICHS_Close(

riichs_info_t * p_riichs_info /* Structure data */

)

Parameters

**p_riichs_info*

This is the pointer to the I²C communication information structure.
Only the member of the structure used in this function is described here. Refer to 2.9 Parameters for details on the structure.

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIICHS_COMMUNICATION) and when an error has occurred (RIICHS_TMO and RIICHS_ERROR).

For the parameter which has '(to be updated)' in the comment below, the argument for the parameter will be updated during the API execution.

```
riichs_ch_dev_status_t dev_sts; /* Device state flag (to be updated) */
uint8_t ch_no;  /* Channel number */
```

Return Values

RIICHS_SUCCESS */* Processing completed successfully */*
RIICHS_ERR_INVALID_CHAN */* The channel is nonexistent. */*
RIICHS_ERR_INVALID_ARG */* Invalid parameter */*

Properties

Prototyped in r_riichs_rx_if.h.

Description

Configures the settings to complete the RIICHS communication. Disables the RIICHS channel specified by the parameter. The following processes are performed in this function.

- Entering the RIICHS module-stop state
- Releasing I²C output ports
- Disabling the RIICHS interrupt

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

Example

```
volatile riichs_return_t  ret;
riichs_info_t             iichs_info_m;

iichs_info_m.ch_no = 0;

ret = R_RIICHS_Close(&iichs_info_m);
```

Special Notes
None

R_RIICHS_GetVersion()

Returns the current version of this module.

Format

uint32_t R_RIICHS_GetVersion(void)

Parameters

None

Return Values

Version number

Properties

Prototyped in r_riichs_rx_if.h.

Description

This function will return the version of the currently installed RIICHS FIT module. The version number is encoded where the top 2 bytes are the major version number and the bottom 2 bytes are the minor version number. For example, Version 4.25 would be returned as 0x00040019.

Example

```
uint32_t    version;

version = R_RIICHS_GetVersion();
```

Special Notes

None.

4. Pin Settings

To use the RIICHS FIT module, assign input/output signals of the peripheral function to pins with the multi-function pin controller (MPC). The pin assignment is referred to as the “Pin Setting” in this document.

The RIICHS FIT module can choose whether or not to perform the pin setting in the R_RIICHS_Open function depending on the setting of the configuration option RIICHS_CFG_PORT_SET_PROCESSING.

For details of the configuration options, refer to "2.7 Configuration Overview".

When performing the Pin Setting in the e² studio, the Pin Setting feature of the FIT Configurator or the Smart Configurator can be used. When using the pin setting feature, pins selected in the Pin Setting pane can be used in the FIT Configurator or Smart Configurator. The information of selected pins is reflected in the r_riichs_pin_config.h file. Values of the macro definitions listed in Table 4.1 are overwritten with values corresponding to the pins selected. When using the pin setting feature of the FIT Configurator, the source file which has the function to enable the pin setting feature (and the "r_pincfg" folder) is not generated in the RIICHS FIT module.

Table 4.1 Macro Definitions for the Pin Setting Feature

Channel Selected	Pin Selected	Macro Definition
Channel 0	SCL0 Pin	R_RIICHS_CFG_RIICHS0_SCL0_PORT R_RIICHS_CFG_RIICHS0_SCL0_BIT
	SDA0 Pin	R_RIICHS_CFG_RIICHS0_SDA0_PORT R_RIICHS_CFG_RIICHS0_SDA0_BIT

Pins selected in the r_riichs_pin_config.h file are configured as peripheral function pins SCL and SDA after calling the R_RIICHS_Open function.

The pins assigned to the peripheral function are released upon calling the R_RIICHS_Close function and then become general I/O pins (as input pins).

Pins SCL and SDA must be pulled up with an external resistor.

When the pin setting feature in this FIT module is not used according to the RIICHS_CFG_PORT_SET_PROCESSING setting, pins used in user processing must be configured after calling the R_RIICHS_Open function before calling the other APIs.

5. Demo Projects

Demo projects are complete stand-alone programs. They include function main() that utilizes the module and its dependent modules (e.g.. r_bsp).

In this section, it explains about GUI operation when you use e² studio.

5.1 riichs_mastersend_demo_rskrx671

Description

A simple demo of the RX671 RIICHS Master Transmission for the RSKRX671 starter kit (FIT module "r_riichs_rx"). The demo uses the RIICHS API from r_riichs_rx_if.h to start master transmission. The master device (RX MCU) transmits data to the slave device. When the master transmission is finished, print the finished message to the debug console by main().

Setup and Execution

1. Compile and download the sample code.
2. Click ‘Reset Go’ to start the software. If PC stops at Main, press F8 to resume.
3. Set breakpoints and watch global variables

Boards Supported

RSKRX671

5.2 riichs_masterreceive_demo_rskrx671

Description

A simple demo of the RX671 RIICHS Master Reception for the RSKRX671 starter kit (FIT module "r_riichs_rx"). The demo uses the RIICHS API from r_riichs_rx_if.h to start master reception. The master (RX MCU) receives data from the slave device .When the master reception is finished, print the received data to the debug console by main().

Boards Supported

RSKRX671

5.3 riichs_slavetransfer_demo_rskrx671

Description

A simple demo of the RX671 RIICHS Slave Transmission and Reception for the RSKRX671 starter kit (FIT module "r_riichs_rx"). The demo uses the RIICHS API from r_riichs_rx_if.h to start slave transmission and reception. The slave (RX MCU) receives data transmitted from the master, or transmits data by the transmit request from the master. When the slave transmission and reception is finished, print the finished message to the debug console by main().

Boards Supported

RSKRX671

5.4 Adding a Demo to a Workspace

Demo projects are found in the FITDemos subdirectory of the distribution file for this application note. To add a demo project to a workspace, select File>Import>General>Existing Projects into Workspace, then click “Next”. From the Import Projects dialog, choose the “Select archive file” radio button. “Browse” to the FITDemos subdirectory, select the desired demo zip file, then click “Finish”.

5.5 Downloading Demo Projects

Demo projects are not included in the RX Driver Package. When using the demo project, the FIT module needs to be downloaded. To download the FIT module, right click on the required application note and select “Sample Code (download)” from the context menu in the Smart Brower >> Application Notes tab.

6. Appendices

6.1 Communication Method

This module controls each processing such as start condition generation, slave address transmission, and others as a single protocol, and performs communication by combining these protocols.

6.1.1 States for API Operation

Table 6.1 lists the States Used for Protocol Control.

Table 6.1 States Used for Protocol Control (enum r_riichs_api_status_t)

No.	Constant Name	Description
STS0	RIICHS_STS_NO_INIT	Uninitialized state
STS1	RIICHS_STS_IDLE	Idle state (ready for master communication)
STS2	RIICHS_STS_IDLE_EN_SLV	Idle state (ready for master/slave communication)
STS3	RIICHS_STS_ST_COND_WAIT	Wait state for a start condition to be detected
STS4	RIICHS_STS_MASTER_CODE_WAIT	Wait state for the Hs mode master code transmission to complete
STS5	RIICHS_STS_SEND_SLVADR_W_WAIT	Wait state for the slave address [write] transmission to complete
STS6	RIICHS_STS_SEND_SLVADR_R_WAIT	Wait state for the slave address [read] transmission to complete
STS7	RIICHS_STS_SEND_DATA_WAIT	Wait state for the data transmission to complete
STS8	RIICHS_STS_RECEIVE_DATA_WAIT	Wait state for the data reception to complete
STS9	RIICHS_STS_SP_COND_WAIT	Wait state for a stop condition to be detected
STS10	RIICHS_STS_AL	Arbitration-lost state
STS11	RIICHS_STS_TMO	Timeout detection state

6.1.2 Events During API Operation

Table 6.2 lists the Events Used for Protocol Control. In this module, not only interrupt but also the module function call is defined as event.

Table 6.2 Events Used for Protocol Control (enum r_riichs_api_event_t)

No.	Event	Event Definition
EV0	RIICHS_EV_INIT	R_RIICHS_Open() called
EV1	RIICHS_EV_EN_SLV_TRANSFER	R_RIICHS_SlaveTransfer() called
EV2	RIICHS_EV_GEN_START_COND	R_RIICHS_MasterSend() or R_RIICHS_MasterReceive() called
EV3	RIICHS_EV_INT_START	EEI interrupt occurred (interrupt flag: START) TEI interrupt occurred (Send master code (0000 1XXXb)) EEI interrupt occurred (interrupt flag: NACK)
EV4	RIICHS_EV_INT_ADD	TEI interrupt occurred, TXI interrupt occurred (1)
EV5	RIICHS_EV_INT_SEND	TEI interrupt occurred, TXI interrupt occurred (1)
EV6	RIICHS_EV_INT_RECEIVE	RXI interrupt occurred
EV7	RIICHS_EV_INT_STOP	EEI interrupt occurred (interrupt flag: STOP)
EV8	RIICHS_EV_INT_AL	EEI interrupt occurred (interrupt flag: AL)
EV9	RIICHS_EV_INT_NACK	EEI interrupt occurred (interrupt flag: NACK)
EV10	RIICHS_EV_INT_TMO	EEI interrupt occurred (interrupt flag: TMO)

Note:
1. The definition of EV4 and EV5 differs depending on the communication operation and the states of "6.1.1 States for API Operation". For details, refer to "6.1.3 Protocol State Transitions".

6.1.3 Protocol State Transitions

In this module, a state transition occurs when an interface function provided is called or when an I²C interrupt request is generated. Figure 6.1 to Figure 6.4 show protocol state transitions.

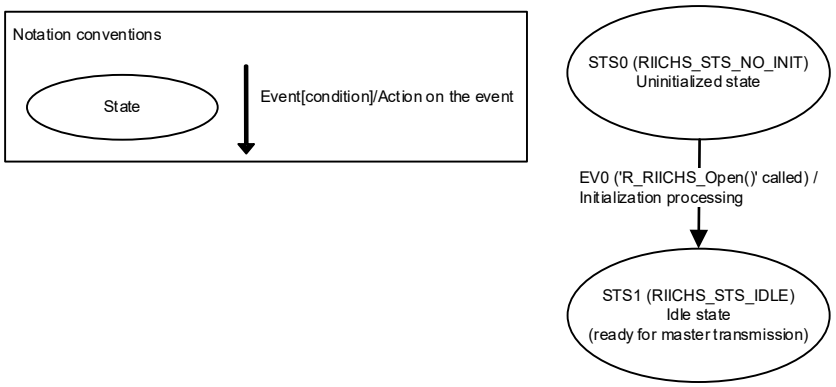


Figure 6.1 State Transition on Initialization (‘R_RIICHS_Open()’ Called)

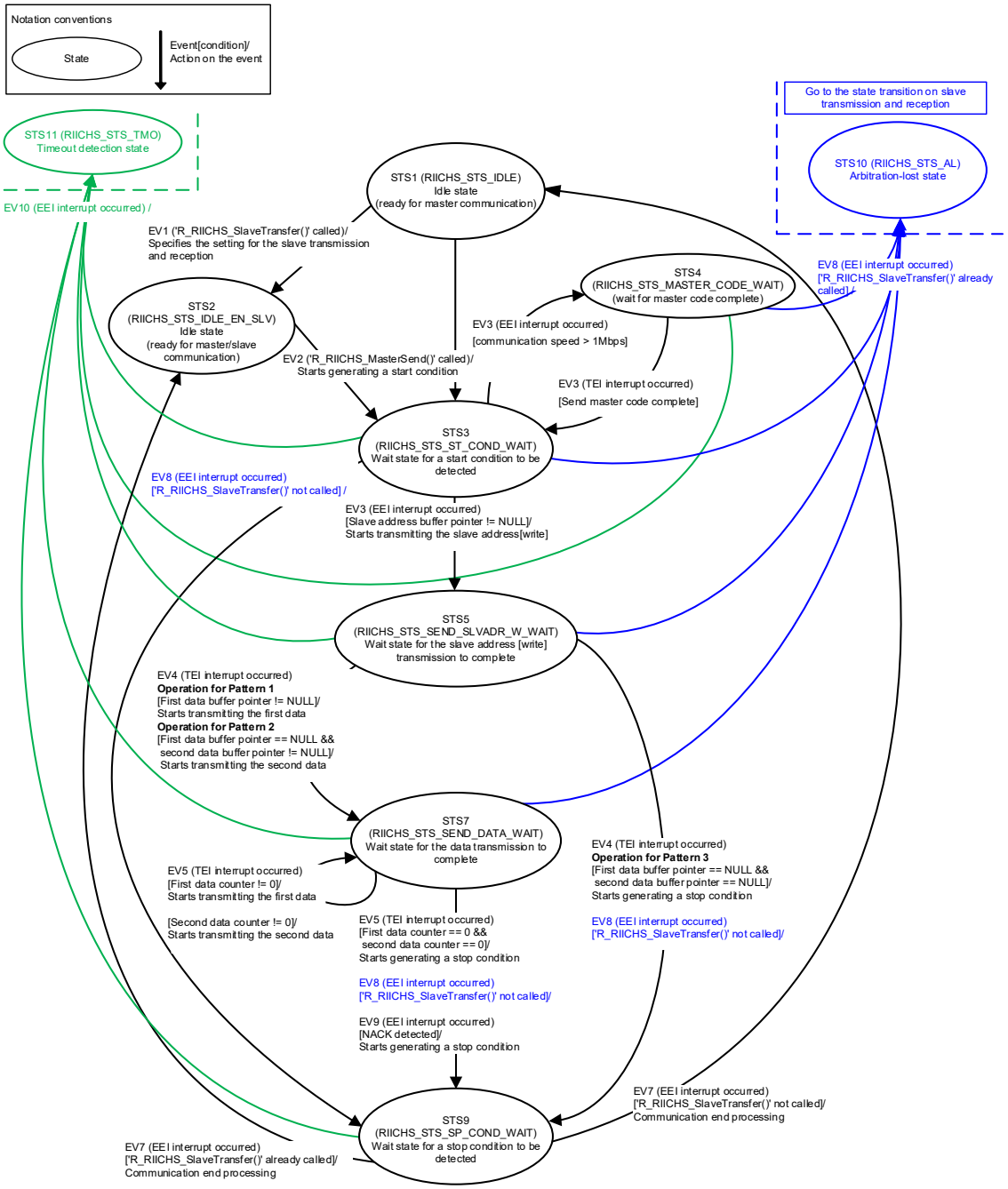


Figure 6.2 State Transition on Master Transmission (R_RIICHS_MasterSend() Called)

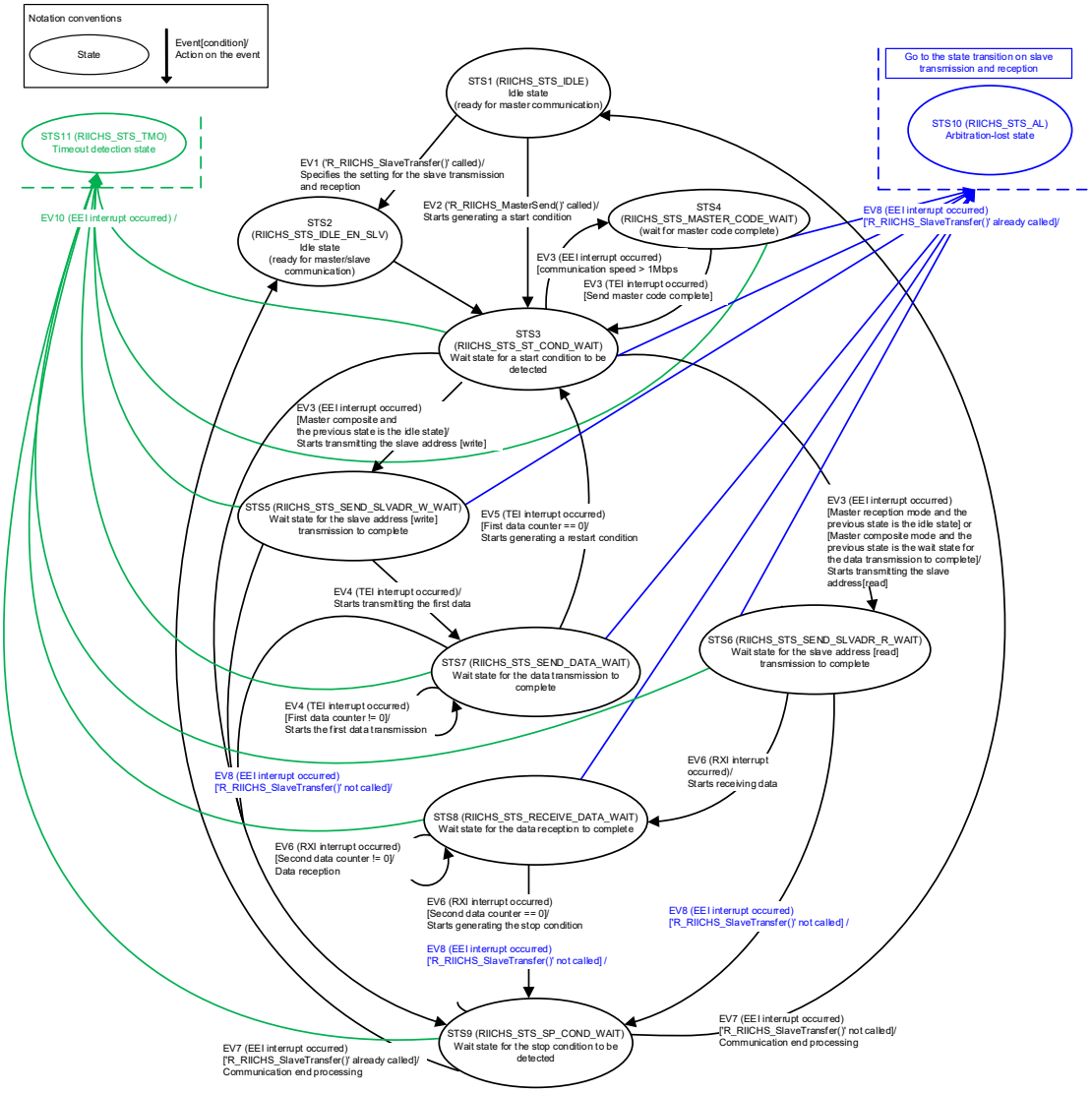


Figure 6.3 State Transition on Master Reception (R_RIICHS_MasterReceive() Called)

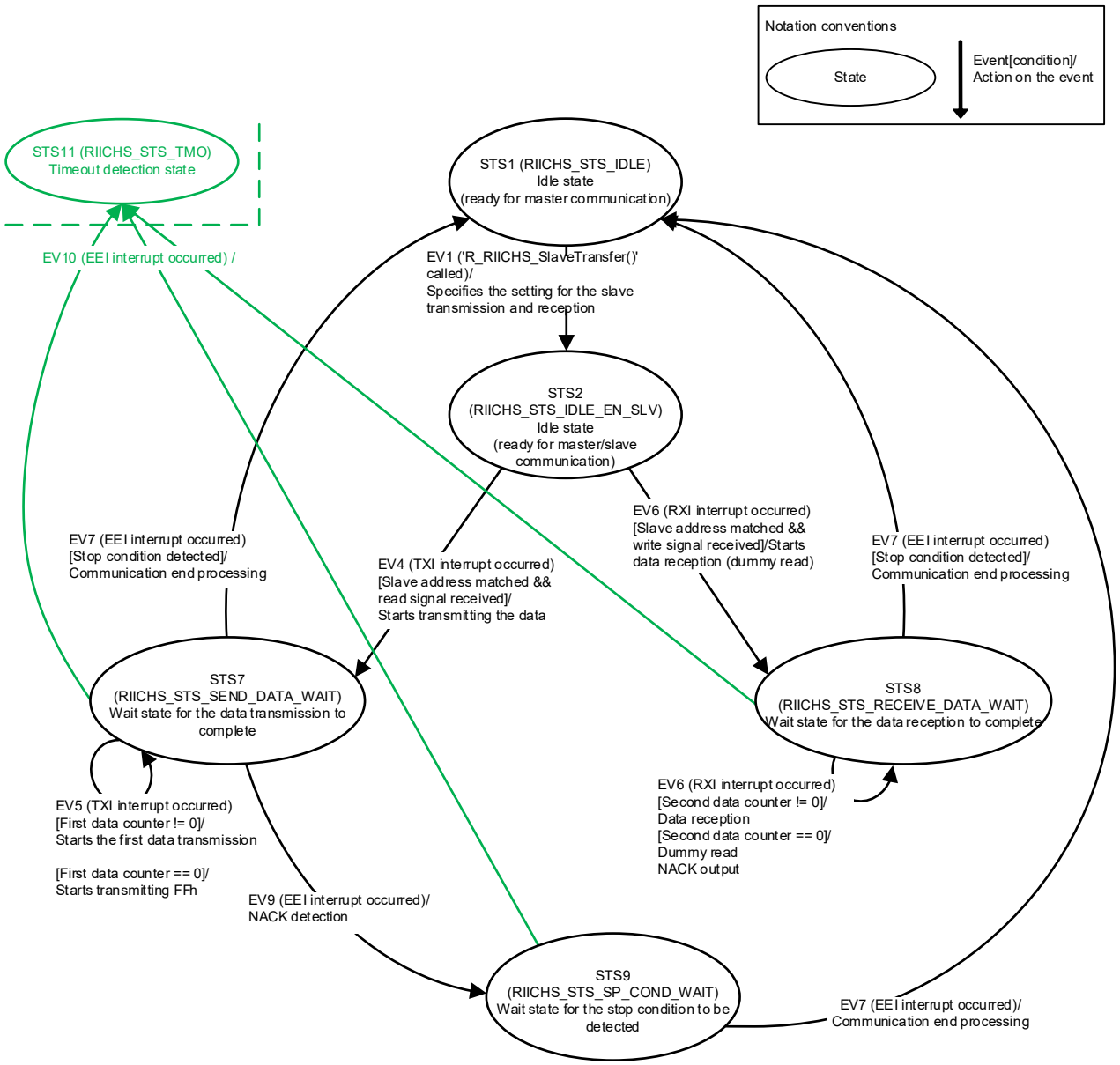


Figure 6.4 State Transition on Slave Transmission and Reception (R_RIICHS_SlaveTransfer() Called)

6.1.4 Protocol State Transition Table

The processing when the events in Table 6.2 occur in the states in Table 6.1 is shown in the Table 6.3 Protocol State Transition. Refer to Table 6.4 for details of each function.

Table 6.3 Protocol State Transition Table (gc_riichs_mtx_tbl[][]) (1)

State		Event										
		EV0	EV1	EV2	EV3	EV4	EV5	EV6	EV7	EV8	EV9	EV10
STS0	Uninitialized state [RIICHS_STS_NO_INIT]	Func0	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
STS1	Idle state (ready for master communication) [RIICHS_STS_IDLE]	ERR	Func 10	Func1	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
STS2	Idle state (ready for master/slave communication) [RIICHS_STS_IDLE_EN_SLV]	ERR	ERR	Func1	ERR	Func4	ERR	Func4	ERR	ERR	ERR	ERR
STS3	Wait state for the start condition to be generated [RIICHS_STS_ST_COND_WAIT]	ERR	ERR	ERR	Func2	ERR	ERR	ERR	ERR	Func8	Func9	Fun c 11
STS4	Wait state for the Hs mode master code transmission to complete [RIICHS_STS_MASTER_CODE_WAIT]	ERR	ERR	ERR	Func 12	ERR	ERR	ERR	ERR	Func8	ERR	Func 11
STS5	Wait state for the slave address [write] to complete [RIICHS_STS_SEND_SLVADR_W_WAIT]	ERR	ERR	ERR	ERR	Func3	ERR	ERR	ERR	Func8	Func9	Func 11
STS6	Wait state for the slave address [read] to complete [RIICHS_STS_SEND_SLVADR_R_WAIT]	ERR	ERR	ERR	ERR	ERR	ERR	Func3	ERR	Func8	Func9	Func 11
STS7	Wait state for the data transmission to complete [RIICHS_STS_SEND_DATA_WAIT]	ERR	ERR	ERR	ERR	ERR	Func5	ERR	ERR	Func8	Func9	Func 11
STS8	Wait state for the data reception to complete [RIICHS_STS_RECEIVE_DATA_WAIT]	ERR	ERR	ERR	ERR	ERR	ERR	Func6	ERR	Func8	Func9	Func 11
STS9	Wait state for the stop condition to be generated [RIICHS_STS_SP_COND_WAIT]	ERR	ERR	ERR	ERR	ERR	ERR	ERR	Func7	ERR	Func9	Func 11
STS10	Arbitration-lost state [RIICHS_STS_AL]	ERR	ERR	ERR	ERR	ERR	Func5	Func6	Func7	ERR	ERR	ERR
STS11	Timeout detection state [RIICHS_STS_TMO]	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR

Note:
1. ERR indicates RIICHS_ERR_OTHER. When an unexpected event is notified in a state, error processing will be performed.

6.1.5 Functions Used on Protocol State Transitions

Table 6.4 lists the Functions Used on Protocol State Transition.

Table 6.4 Functions Used on Protocol State Transition

Processing	Function	Overview
Func0	riichs_init_driver()	Initialization
Func1	riichs_generate_start_cond()	Start condition generation (for master transmission)
Func2	riichs_after_gen_start_cond()	Processing after generating a start condition
Func3	riichs_after_send_slvadr()	Processing after completing the slave address transmission
Func4	riichs_after_receive_slvadr()	Processing after matching the received slave address
Func5	riichs_write_data_sending()	Data transmission
Func6	riichs_read_data_receiving()	Data reception
Func7	riichs_after_dtct_stop_cond ()	Communication end processing
Func8	riichs_arbitration_lost()	Processing when detecting an arbitration-lost
Func9	riichs_nack()	Processing when detecting a NACK
Func10	riichs_enable_slave_transfer()	Enabling slave transmission/reception
Func11	riichs_time_out()	Processing when detecting a timeout
Func12	riichs_send_master_code_cond()	Master code (0000 1XXXb) transmission

6.1.6 Flag States on State Transitions

1. Controlling states of channels

Multiple slaves on the same bus can be exclusively controlled using the channel state flag ‘g_riichs_ChStatus[]’. Each channel has the channel state flag and the flag is controlled by the global variable. When the initialization for this module has completed and the target bus is not being used for a communication, the flag becomes ‘RIICHS_IDLE/RIICHS_FINISH/RIICHS_NACK’ (idle state (ready for communication)) and communication is available. When the bus is being used for communication, the flag becomes ‘RIICHS_COMMUNICATION’ (communicating). When communication is started, the flag is always verified. Thus, if a device is communicating on a bus, then no other device can start communicating on the same bus. Simultaneous communication can be achieved by controlling the channel state flag for each channel.

2. Controlling states of devices

Multiple slaves on the same channel can be controlled using the device state flag ‘dev_sts’ in the I²C communication information structure. The device state flag stores the state of communication for the device.

Table 6.5 lists States of Flags on State Transitions.

Table 6.5 States of Flags on State Transitions

State	Channel State Flag	Device State Flag (Communication Device)	I²C Protocol Operating Mode	Current State of the Protocol Control
	g_riichs_ChStatus[]	I²C Communication Information Structure dev_sts	Internal Communication Information Structure N_Mode	Internal Communication Information Structure N_status
Uninitialized state	RIICHS_NO_INIT	RIICHS_NO_INIT	RIICHS_MODE_NONE	RIICHS_STS_NO_INIT
Idle state (ready for master communication)	RIICHS_IDLE RIICHS_FINISH RIICHS_NACK	RIICHS_IDLE RIICHS_FINISH RIICHS_NACK	RIICHS_MODE_NONE	RIICHS_STS_IDLE
Idle state (ready for master/slave communication)	RIICHS_IDLE	RIICHS_IDLE	RIICHS_MODE_S_READY	RIICHS_STS_IDLE_EN_SLV
Communicating (master transmission)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_M_SEND	RIICHS_STS_ST_COND_WAIT
				RIICHS_STS_MASTER_CODE_WAIT
				RIICHS_STS_SEND_SLVADR_W_WAIT
				RIICHS_STS_SEND_DATA_WAIT
				RIICHS_STS_SP_COND_WAIT
				RIICHS_STS_AL
				RIICHS_STS_TMO
Communicating (master reception)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_M_RECEIVE	RIICHS_STS_ST_COND_WAIT
				RIICHS_STS_MASTER_CODE_WAIT
				RIICHS_STS_SEND_SLVADR_R_WAIT
				RIICHS_STS_RECEIVE_DATA_WAIT
				RIICHS_STS_SP_COND_WAIT
				RIICHS_STS_AL
				RIICHS_STS_TMO
Communicating (master transmit/receive)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_M_SEND_RECEIVE	RIICHS_STS_ST_COND_WAIT
				RIICHS_STS_SEND_SLVADR_W_WAIT
				RIICHS_STS_SEND_SLVADR_R_WAIT
				RIICHS_STS_SEND_DATA_WAIT
				RIICHS_STS_RECEIVE_DATA_WAIT
				RIICHS_STS_SP_COND_WAIT
				RIICHS_STS_AL
				RIICHS_STS_TMO
Communicating (slave transmission)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_S_SEND	RIICHS_STS_SEND_DATA_WAIT
				RIICHS_STS_SP_COND_WAIT
				RIICHS_STS_TMO
Communicating (slave reception)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_S_RECEIVE	RIICHS_STS_RECEIVE_DATA_WAIT
				RIICHS_STS_SP_COND_WAIT
				RIICHS_STS_TMO
Arbitration-lost detection state	RIICHS_AL	RIICHS_AL	—	—
Timeout detection state	RIICHS_TMO	RIICHS_TMO	—	—
Error state	RIICHS_ERROR	RIICHS_ERROR	—	—

6.2 Interrupt Request Generation Timing

This section describes the interrupt request generation timings in this module.

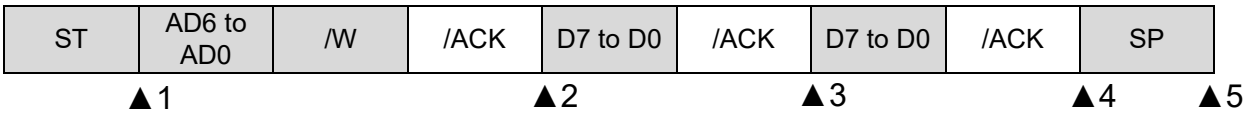
Legend:
ST: Start condition
AD6 to AD0: Slave address
/W: Transfer direction bit: 0 (Write)
R: Transfer direction bit: 1 (Read)
/ACK: Acknowledge: 0
NACK: Acknowledge: 1
D7 to D0: Data
RST: Restart condition
SP: Stop condition

From master to slave

From slave to master

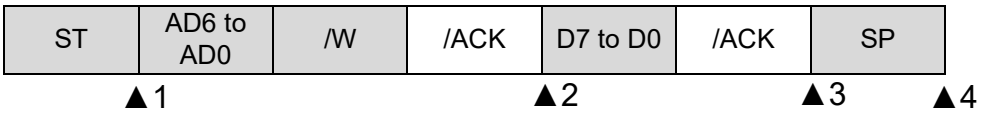
6.2.1 Master Transmission

(1) Pattern 1



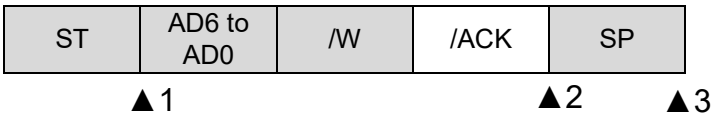
▲1: EEI (START) interrupt: Start condition detected
▲2: TEI interrupt: Address transmission completed (transfer direction bit: write)
▲3: TEI interrupt: Data transmission completed (first data)
▲4: TEI interrupt: Data transmission completed (second data)
▲5: EEI (STOP) interrupt: Stop condition detected

(2) Pattern 2



▲1: EEI (START) interrupt: Start condition detected
▲2: TEI interrupt: Address transmission completed (transfer direction bit: write)
▲3: TEI interrupt: Data transmission completed (second data)
▲4: EEI (STOP) interrupt: Stop condition detected

(3) Pattern 3

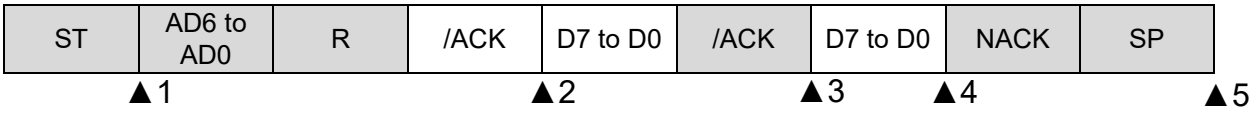


▲1: EEI (START) interrupt: Start condition detected

▲2: TEI interrupt: Address transmission completed (transfer direction bit: write)

▲3: EEI (STOP) interrupt: Stop condition detected

6.2.2 Master Reception



▲1: EEI (START) interrupt: Start condition detected

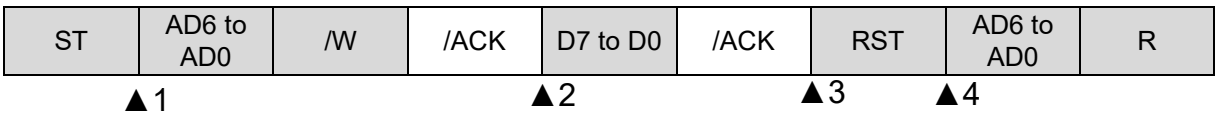
▲2: RXI interrupt: Address transmission completed (transfer direction bit: read)

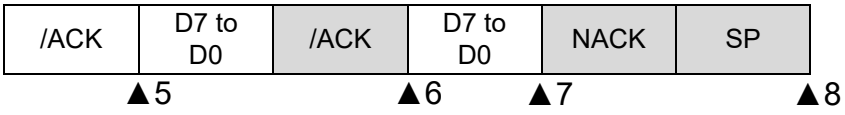
▲3: RXI interrupt: Reception for the last data - 1 completed (second data)

▲4: RXI interrupt: Reception for the last data completed (second data)

▲5: EEI (STOP) interrupt: Stop condition detected

6.2.3 Master Transmit/Receive





▲1: EEI (START) interrupt: Start condition detected

▲2: TEI interrupt: Address transmission completed (transfer direction bit: write)

▲3: TEI interrupt: Data transmission completed (first data)

▲4: EEI (START) interrupt: Restart condition detected

▲5: RXI interrupt: Address transmission completed (transfer direction bit: read)

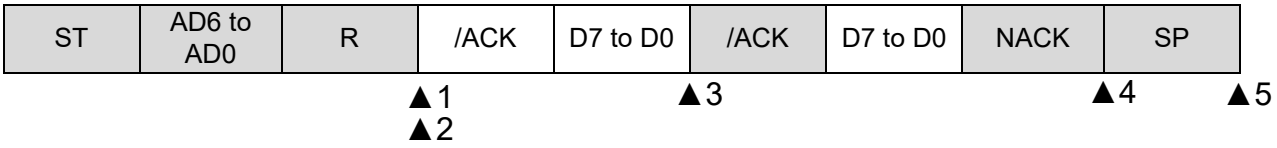
▲6: RXI interrupt: Reception for the last data - 1 completed (second data)

▲7: RXI interrupt: Reception for the last data completed (second data)

▲8: EEI (STOP) interrupt: Stop condition detected

6.2.4 Slave Transmission

When transmitting 2-byte data:



▲1: TXI interrupt: Received address matched (transfer direction bit: read)

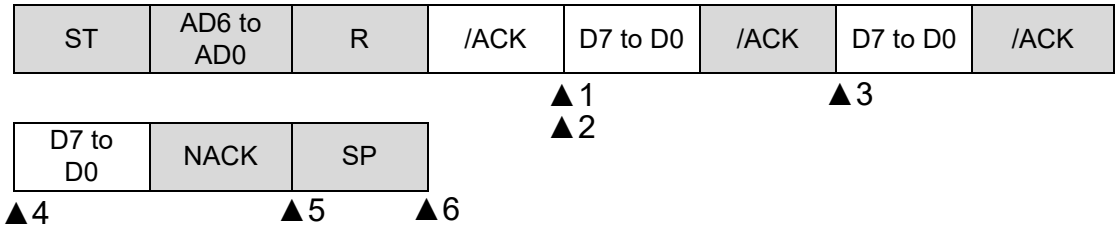
▲2: TXI interrupt: Transmit buffer is empty

▲3: TXI interrupt: Transmit buffer is empty

▲4: EEI (NACK) interrupt: NACK detected

▲5: EEI (STOP) interrupt: Stop condition detected

When transmitting 3-byte data:



▲1: TXI interrupt: Received address matched (transfer direction bit: read)

▲2: TXI interrupt: Transmit buffer is empty

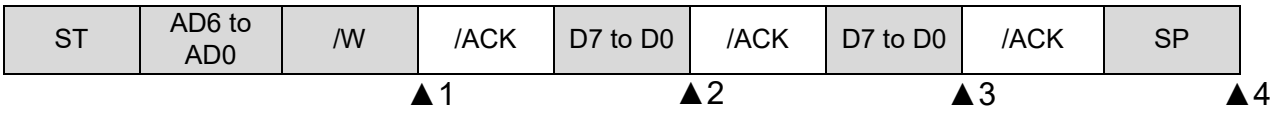
▲3: TXI interrupt: Transmit buffer is empty

▲4: TXI interrupt: Transmit buffer is empty

▲5: EEI (NACK) interrupt: NACK detected

▲6: EEI (STOP) interrupt: Stop condition detected

6.2.5 Slave Reception



▲1: RXI interrupt: Received address matched (transfer direction bit: write)

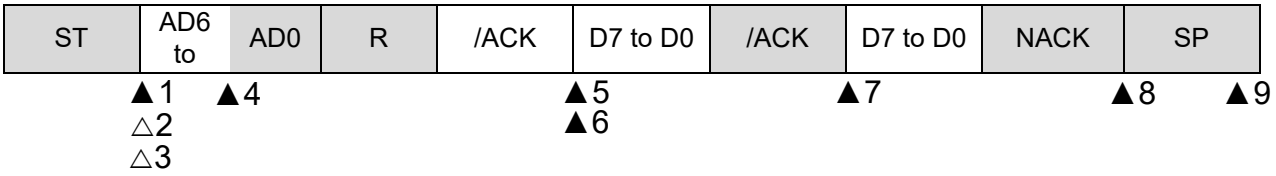
▲2: RXI interrupt: Reception for the last data - 1 completed (second data)

▲3: RXI interrupt: Reception for the last data completed (second data)

▲4: EEI (STOP) interrupt: Stop condition detected

6.2.6 Multi-Master Communication

(Slave transmission after detecting AL during master transmission)



▲1: EEI (START) interrupt: Start condition detected

△ 2: TXI interrupt: Start condition detected (no processing performed)

△ 3: TXI interrupt: Transmit buffer is empty (no processing performed)

▲4: EEI (AL) interrupt: Arbitration-lost detected

▲5: TXI interrupt: Address reception matched (transfer direction bit: Read)

▲6: TXI interrupt: Transmit buffer is empty

▲7: TXI interrupt: Transmit buffer is empty

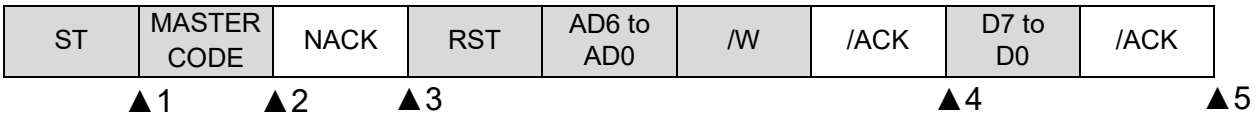
▲8: EEI (NACK) interrupt: NACK detected

▲9: EEI (STOP) interrupt: Stop condition detected

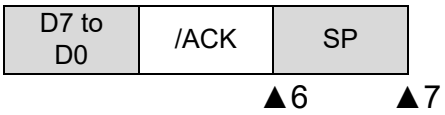
6.2.7 Master Transmission in Hs mode

(1) Pattern 1

|<----- Except in Hs mode --------->|<----- Hs mode----------->|



|<-----Hs mode ----------- ----->|<----- Except in Hs mode --->|



▲1: EEI (START) interrupt: Start condition detected

▲2: TEI interrupt: Master code (0000 1XXXb) transmission completed

▲3: EEI (NACK) interrupt: NACK detected

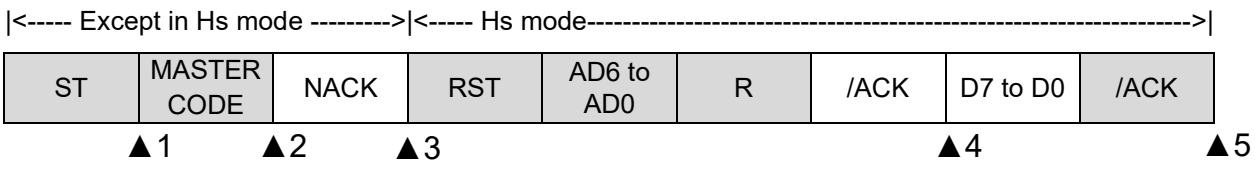
▲4: TEI interrupt: Address transmission completed (transfer direction bit: write)

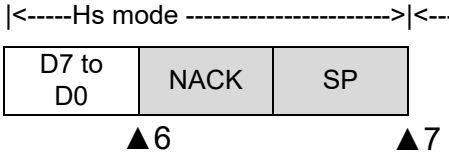
▲5: TEI interrupt: Data transmission completed (first data)

▲6: TEI interrupt: Data transmission completed (second data)

▲7: EEI (STOP) interrupt: Stop condition detected

6.2.8 Master Reception in Hs mode





▲1: EEI (START) interrupt: Start condition detected

▲2: TEI interrupt: 8-bit master code (0000 1XXX) transmission completed

▲3: EEI (NACK) interrupt: NACK detected

▲4: RXI interrupt: Address transmission completed (transfer direction bit: read)

▲5: RXI interrupt: Reception for the last data - 1 completed (second data)

▲6: RXI interrupt: Reception for the last data completed (second data)

▲7: EEI (STOP) interrupt: Stop condition detected

6.3 Timeout Detection and Processing After the Detection

6.3.1 Detecting a Timeout with the Timeout Detection Function

When the timeout detection function is enabled by the argument when executing R_RIICHS_Open() fucntion, call the R_RIICHS_GetStatus() function in the callback function.

The information of timeout detection can be verified with the TMO bit in the riichs_mcu_status_t structure specified as the second parameter in the R_RIICHS_GetStatus() function.

- When the TMO bit is 1: Timeout detected

- When the TMO bit is 0: Timeout not detected

6.3.2 Processing After a Timeout is Detected

When a timeout is detected, the R_RIICHS_Close() function needs to be called once to restart communication calling the R_RIICHS_Open() function in the initialization.

A timeout may be detected due to a bus hang up. In master mode, if the clock signals from the master and slave devices go out of synchronization due to noise or other factors, the slave device may hold the SDA line low (bus hang up). Then the stop condition cannot be issued and a timeout will be detected.

To recover from bus hang up state, the extra SCL clock cycle output function is used. Outputting one clock of the extra SCL at a time can release the SDA line from being held low and the bus is recovered from hang up state.

To output one clock of the extra SCL clock, set “RIICHS_GEN_SCL_ONESHOT” (one-shot output of the SCL clock) to the second parameter of the R_RIICHS_Control() function and call the R_RIICHS_Control() function.

The state of the SCL pin can be verified using the R_RIICHS_GetStatus() function.

Repeat one-shot output of the SCL clock until the SCL clock becomes high.

Figure 6.5 shows the Timeout Detection and Processing After the Detection.

For details on the extra SCL clock cycle output function, refer to the Extra SCL Clock Cycle Output Function section of the High-Speed I²C Bus Interface (RIICHS) chapter in the User’s Manual: Hardware for the product used.

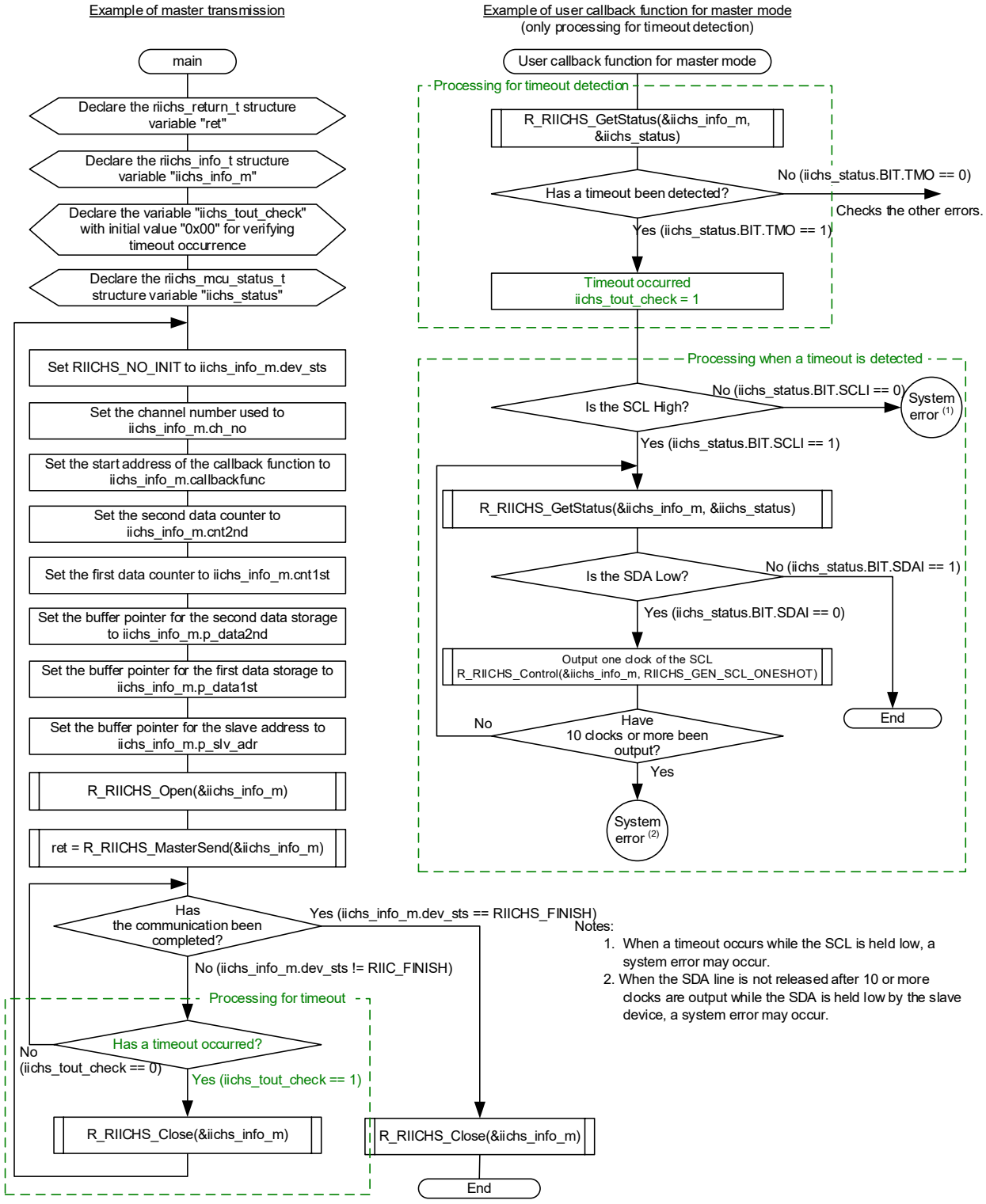


Figure 6.5 Timeout Detection and Processing After the Detection

6.4 Operating Test Environment

This section describes for detailed the operating test environments of this module.

Table 6.6 Operation Confirmation Environment for Rev.1.00.

Item	Contents
Integrated development environment	Renesas Electronics e² studio 2020-10 (20.10.0) IAR Embedded Workbench for Renesas 4.14.01
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V.3.03.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.03.00.202002 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.14.01 Compiler option: The default settings of the integrated development environment.
Endian order	Big-endian/Little-endian
Module version	Rev.1.00
Board used	Renesas Starter Kit+ for RX671 (product number. RTK55671xxxxxxxxxx)

Table 6.7 Operation Confirmation Environment for Rev.1.10.

Item	Contents
Integrated development environment	Renesas Electronics e2 studio 2022-10 (22.10.0) IAR Embedded Workbench for Renesas 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V.3.04.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.03.00.202204 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian order	Big-endian/Little-endian
Module version	Rev.1.10
Board used	Renesas Starter Kit+ for RX671 (product number. RTK55671xxxxxxxxxx)

Table 6.8 Operation Confirmation Environment for Rev.1.20.

Item	Contents
Integrated development environment	Renesas Electronics e2 studio 2024-07 (24.7.0) IAR Embedded Workbench for Renesas 5.10.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V.3.06.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.03.00.202405 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 5.10.1 Compiler option: The default settings of the integrated development environment.
Endian order	Big-endian/Little-endian
Module version	Rev.1.20
Board used	None

Table 6.9 Operation Confirmation Environment for Rev.1.21.

Item	Contents
Integrated development environment	Renesas Electronics e2 studio 2025-01 (25.1.0) IAR Embedded Workbench for Renesas 5.10.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V.3.07.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.03.00.202411 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 5.10.1 Compiler option: The default settings of the integrated development environment.
Endian order	Big-endian/Little-endian
Module version	Rev.1.21
Board used	-

6.5 Troubleshooting

(1) Q: I have added the FIT module to the project and built it. Then I got the error: Could not open source file “platform.h”.

A: The FIT module may not be added to the project properly. Check if the method for adding FIT modules is correct with the following documents:

- When using CS+:

  Application note “Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)”

- When using e² studio:

  Application note “Adding Firmware Integration Technology Modules to Projects (R01AN1723)”

When using a FIT module, the board support package FIT module (BSP module) must also be added to the project. For this, refer to the application note “Board Support Package Module Using Firmware Integration Technology (R01AN1685)”.

(2) Q: I have added the FIT module to the project and built it. Then I got the error: This MCU is not supported by the current r_riichs_rx module.

A: The FIT module you added may not support the target device chosen in the user project. Check if the FIT module supports the target device for the project used.

(3) Q: I have added the FIT module to the project and built it. Then I got an error for when the configuration setting is wrong.

A: The setting in the file “r_riichs_rx_config.h” may be wrong. Check the file “r_riichs_rx_config.h”. If there is a wrong setting, set the correct value for that. Refer to 2.7 Configuration Overview for details.

6.6 Sample Code

6.6.1 Example when Accessing One Slave Device Continuously with One Channel

This section describes an example of using one RIICHS channel to continuously access to one slave device.

The procedure is as follows:

1. Execute the R_RIICHS_Open function to use RIICHS channel 0 in the RIICHS FIT module.
2. Execute the R_RIICHS_MasterSend function to write 16-byte data to EEPROM.
3. Performs Acknowledge Polling to wait for EEPROM write completion.
4. Execute the R_RIICHS_MasterReceive function to write 16-byte data from EEPROM.
5. Compare write data with read data.
6. Execute the R_RIICHS_Close function to release RIICHS channel 0 from the RIICHS FIT module.

This sample code is checked to operate with Renesas starter kit of target device. Please note that the address of the slave device depends on the EEPROM used.

```
#include <stddef.h>
#include "platform.h"
#include "r_riichs_rx_if.h"

/* EEPROM device code (fixed) */
#define    EEPROM_DEVICE_CODE       (0xA0)

/* Device address code(under 4 bit is A2(Vss=0), A1(Vcc=1), A0(Vcc=1), and RW code)
   for hardware connection with EEPROM on RSK of the supported target device.
   Please change the following settings as necessary. */
#define    EEPROM_DEVICE_ADDRESS_CODE    (0x06)

/* E2PROM device address */
#define    EEPROM_DEVICE_ADDRESS    ((EEPROM_DEVICE_CODE | EEPROM_DEVICE_ADDRESS_CODE) >> 1)

/* variables */
static volatile riichs_return_t ret;   /* Return value */
static riichs_info_t iichs_info_m;       /* Structure data */

static uint8_t addr_eeprom[1] = { EEPROM_DEVICE_ADDRESS };
static uint8_t access_addr1[1] = { 0x00 };

/* This data is sent to the EEPROM when target device is the master device. */
static uint8_t master_send_data[16] =
{ 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e,
0x8f };

/* This buffer stores data received from the slave device. */
static uint8_t master_store_area[16] =
{ 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF };

/* private functions */
static void callback_master (void);
static void eeprom_write (void);
static void acknowledge_polling (void);
static void eeprom_read (void);
```

Figure 6.6 Example when Accessing One Slave Device Continuously with One Channel (1/6)

```
/***********************************************************************************************
* Function Name: main
* Description  : The main loop
* Arguments    : none
* Return Value : none
***********************************************************************************************/
void main (void)
{
    uint8_t i = 0;

    /* Initialize */
    for (i = 0; i < 16; i++)
    {
        master_store_area[i] = 0xFF;
    }

    /* Set arguments for R_RIICHS_Open. */
    iichs_info_m.ch_no = 0;              /* Channel number */
    iichs_info_m.dev_sts = RIICHS_NO_INIT; /* Device state flag (to be updated) */
    iichs_info_m.scl_up_time = 20E-9; /* Rise time of SCLn Line */
    iichs_info_m.scl_down_time = 20E-9; /* Fall time of SCLn Line */
    iichs_info_m.fs_scl_up_time = 20E-9; /* Rise time of SCLn Line before transition to Hs mode */
    iichs_info_m.fs_scl_down_time = 20E-9; /* Fall time of SCLn Line before transition to Hs mode
*/
    iichs_info_m.speed_kbps = 3400; /* RIICHS bps(kbps) */
    iichs_info_m.fs_speed_kbps = 400; /* RIICHS bps(kbps) before transition to Hs mode */
    iichs_info_m.bus_check_counter = 1000; /* software bus busy check counter */
    iichs_info_m.bus_free_time = 5; /* software bus free counter*/
    iichs_info_m.slave_addr0 = 0x0025; /* Slave address 0 */
    iichs_info_m.slave_addr1 = 0x0000; /* Slave address 1 */
    iichs_info_m.slave_addr2 = 0x0000; /* Slave address 2 */
    iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT; /* Slave address 0 format */
    iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE; /* Slave address 1 format */
    iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE; /* Slave address 2 format */
    iichs_info_m.gca_enable = RIICHS_GCA_DISABLE; /* Disable General call address */
    iichs_info_m.rxi_priority = RIICHS_IPL_1; /* The priority level of the RXI */
    iichs_info_m.txi_priority = RIICHS_IPL_1; /* The priority level of the TXI */
    iichs_info_m.eei_priority = RIICHS_IPL_1; /* The priority level of the EEI */
    iichs_info_m.tei_priority = RIICHS_IPL_1; /* The priority level of the TEI */
    iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE; /* Disable Master Arbitration-Lost
Detection */
    iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0; /* digital noise filter stage */
    iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE; /* Enable Timeout function */
    iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE; /* Enable NACK Detection */
    iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE; /* Enable Arbitration Lost*/
    iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16; /* 16 bit for the timeout detection time */
    iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE; /* SCL line is held LOW when the timeout
function is enabled */
    iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE; /* SCL line is held HIGH when the timeout
function is enabled */
    iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL; /* Timeout Detection Mode */


    ret = R_RIICHS_Open(&iichs_info_m);
    if (RIICHS_SUCCESS != ret)
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIICHS_SUCCESS'. */
        while (1)
        {
            R_BSP_NOP();    /* error */
        }
    }

    /* EEPROM Write (Master transfer) */
    eeprom_write();

    /* Acknowledge polling (Master transfer) */
    acknowledge_polling();

    /* EEPROM Read (Master transfer and Master receive) */
    eeprom_read();
```

Figure 6.7 Example when Accessing One Slave Device Continuously with One Channel (2/6)

```
    /* Compare */
    for (i = 0; i < 16; i++)
    {
        if (master_store_area[i] != master_send_data[i])
        {
            /* Detected mismatch. */
            LED3 = LED_ON;
        }
        else
        {
            LED0 = LED_ON;
        }
    }

    ret = R_RIICHS_Close(&iichs_info_m);
    if (RIICHS_SUCCESS != ret)
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIICHS_SUCCESS'. */
        while (1)
        {
            R_BSP_NOP();    /* error */
        }
    }

    while (1)
    {
        /* do nothing */
    }

} /* End of function main() */
```

Figure 6.8 Example when Accessing One Slave Device Continuously with One Channel (3/6)

```
/***********************************************************************************************
* Function Name: callback_master
* Description  : This function is sample of Master Mode callback function.
* Arguments    : none
* Return Value : none
***********************************************************************************************/
static void callback_master (void)
{
    riichs_mcu_status_t      iichs_status;

    ret = R_RIICHS_GetStatus(&iichs_info_m, &iichs_status);
    if (RIICHS_SUCCESS != ret)
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIICHS_SUCCESS'. */
        while (1)
        {
            R_BSP_NOP();    /* error */
        }
    }
    else
    {
        /* Processing when a timeout, arbitration-lost, NACK,
           or others is detected by verifying the iichs_status flag. */
    }

} /* End of function callback_master() */

/***********************************************************************************************
* Function Name: eeprom_write
* Description  : This function is sample of EEPROM write function using R_RIICHS_MasterSend.
* Arguments    : none
* Return Value : none
***********************************************************************************************/
static void eeprom_write (void)
{
    /* Set arguments for R_RIICHS_MasterSend. */
    iichs_info_m.p_slv_adr = addr_eeprom;  /* Pointer to the slave address storage buffer */
    iichs_info_m.p_data1st = access_addr1; /* Pointer to the first data storage buffer */
    iichs_info_m.cnt1st = 1;               /* First data counter (number of bytes)(to be updated)
*/
    iichs_info_m.p_data2nd = master_send_data;    /* Pointer to the second data storage buffer */
    iichs_info_m.cnt2nd = 16;           /* Second data counter (number of bytes)(to be updated) */
    iichs_info_m.callbackfunc = &callback_master; /* Callback function */

    /* Master send start */
    ret = R_RIICHS_MasterSend(&iichs_info_m);
    if (RIICHS_SUCCESS == ret)
    {
        /* Waiting for R_RIICHS_MasterSend completed. */
        while (RIICHS_COMMUNICATION == iichs_info_m.dev_sts)
        {
            /* do nothing */
        }

        if (RIICHS_NACK == iichs_info_m.dev_sts)
        {
            /* Slave returns NACK. The slave address may not correct.
               Please check the macro definition value or hardware connection etc. */
            while (1)
            {
                R_BSP_NOP();    /* error */
            }
        }
    }
    else
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIICHS_SUCCESS'. */
```

Figure 6.9 Example when Accessing One Slave Device Continuously with One Channel (4/6)

```
        while (1)
        {
            R_BSP_NOP();    /* error */
        }
    }

} /* End of function eeprom_write() */


/***********************************************************************************************
* Function Name: acknowledge_polling
* Description  : This function is sample of Acknowledge Polling using R_RIICHS_MasterSend with
*                master send pattern 3.
* Arguments    : none
* Return Value : none
***********************************************************************************************/
static void acknowledge_polling (void)
{
    do
    {
        /* Set arguments for R_RIICHS_MasterSend. */
        iichs_info_m.p_slv_adr = addr_eeprom;      /* Pointer to the slave address storage buffer
*/
        iichs_info_m.p_data1st = (uint8_t*) FIT_NO_PTR; /* Pointer to the first data storage
buffer */
        iichs_info_m.cnt1st = 0;                        /* First data counter (number of bytes) */
        iichs_info_m.p_data2nd = (uint8_t*) FIT_NO_PTR; /* Pointer to the second data storage
buffer */
        iichs_info_m.cnt2nd = 0;                        /* Second data counter (number of bytes)
*/
        iichs_info_m.callbackfunc = &callback_master;   /* Callback function */

        /* Master send start. */
        ret = R_RIICHS_MasterSend(&iichs_info_m);
        if (RIICHS_SUCCESS == ret)
        {
            /* Waiting for R_RIICHS_MasterSend completed. */
            while (RIICHS_COMMUNICATION == iichs_info_m.dev_sts)
            {
                /* do nothing */
            }

            /* Slave returns NACK. Set retry interval. */
            if (RIICHS_NACK == iichs_info_m.dev_sts)
            {
                /* Waiting for retry interval 100us. */
                R_BSP_SoftwareDelay(100, BSP_DELAY_MICROSECS);
            }
        }
        else
        {
            /* This software is for single master.
               Therefore, return value should be always 'RIICHS_SUCCESS'. */
            while (1)
            {
                R_BSP_NOP();    /* error */
            }
        }
    } while (RIICHS_FINISH != iichs_info_m.dev_sts);

} /* End of function acknowledge_polling() */
```

Figure 6.10 Example when Accessing One Slave Device Continuously with One Channel (5/6)

```
/***********************************************************************************************
* Function Name: eeprom_read
* Description  : This function is sample of EEPROM read function using R_RIICHS_MasterReceive.
* Arguments    : none
* Return Value : none
***********************************************************************************************/
static void eeprom_read (void)
{
    /* Set arguments for R_RIICHS_MasterReceive. */
    iichs_info_m.p_slv_adr = addr_eeprom;         /* Pointer to the slave address storage buffer
*/
    iichs_info_m.p_data1st = access_addr1;        /* Pointer to the first data storage buffer */
    iichs_info_m.cnt1st = 1;              /* First data counter (number of bytes)(to be updated)
*/
    iichs_info_m.p_data2nd = master_store_area;   /* Pointer to the second data storage buffer */
    iichs_info_m.cnt2nd = 16;             /* Second data counter (number of bytes)(to be updated)
*/
    iichs_info_m.callbackfunc = &callback_master; /* Callback function */

    /* Master send receive start. */
    ret = R_RIICHS_MasterReceive(&iichs_info_m);
    if (RIICHS_SUCCESS == ret)
    {
        /* Waiting for R_RIICHS_MasterSend completed. */
        while (RIICHS_COMMUNICATION == iichs_info_m.dev_sts)
        {
            /* do nothing */
        }

        if (RIICHS_NACK == iichs_info_m.dev_sts)
        {
            /* Slave returns NACK. The slave address may not correct.
               Please check the macro definition value or hardware connection etc. */
            while (1)
            {
                R_BSP_NOP();    /* error */
            }
        }
    }
    else
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIICHS_SUCCESS'. */
        while (1)
        {
                R_BSP_NOP();    /* error */
        }
    }

} /* End of function eeprom_read() */
```

Figure 6.11 Example when Accessing One Slave Device Continuously with One Channel (6/6)

7. Reference Documents

User's Manual: Hardware
The latest version can be downloaded from the Renesas Electronics website.

Technical Update/Technical News
The latest information can be downloaded from the Renesas Electronics website.

User's Manual: Development Tools
RX Family Compiler CC-RX User's Manual (R20UT3248)
The latest versions can be downloaded from the Renesas Electronics website.

Related Technical Updates

This module reflects the content of the following technical updates.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jun. 30, 2021	-	First edition issued
1.10	Dec. 21, 2022	71 Program	6.4 Operating Test Environment: Added Table for Rev. 1.10. Fixed processing error of riichs_bps_calc
1.20	Nov. 01, 2024	73 Program	6.4 Operating Test Environment: Added Table for Rev. 1.20. Changed the comment of API functions to the doxygen style.
1.21	Mar. 15, 2025	21	2.3 Supported Toolchains Added for Toolchain v.3.07.00.
		73	6.4 Operating Test Environment: Added Table for Rev. 1.21.
		Program	Updated FIT Disclaimer and Copyright.

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

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

1. Precaution against Electrostatic Discharge (ESD)
   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on
   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state
   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins
   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals
   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses
   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products
   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact information

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

Trademarks

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