RX Family

R01AN1691EJ0220
Rev. 2.20
Aug. 31, 2017

Simple I²C Module Using Firmware Integration Technology

Introduction

This application note describes the simple I²C module using firmware integration technology (FIT) for communications between devices using the serial communications interface (SCI).

Target Device

This API supports the following device.

- RX110, RX111, RX113 Groups
- RX130 Group
- RX230, RX231, RX23T Groups
- RX24T, RX24U Groups
- RX63N Group
- RX64M Group
- RX65N, RX651 Groups
- RX71M Group

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

Related Documents

- Firmware Integration Technology User's Manual (R01AN1833)
- Board Support Package Module Using Firmware Integration Technology (R01AN1685)
- Adding Firmware Integration Technology Modules to Projects (R01AN1723)
- Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)
- Renesas e² studio Smart Configurator User Guide(R20AN0451)

Contents

1. Overview

The simple I2C module using firmware integration technology (SCI simple I2C mode FIT module [1]) provides a method to transmit and receive data between the master and slave devices using the SCI. The SCI simple I2C mode is in compliance with single master mode of the NXP I2C-bus (Inter-IC-Bus) interface.

Note:

1. When the description says "module" in this document, it indicates the SCI simple I2C mode FIT module.

Features supported by this module are as follows:

- Single master mode (slave transmission or slave reception is not supported).

- Bus condition waveform generation

- Communication mode can be standard or fast mode and the maximum communication rate is 384 kbps.

Limitations

- This module cannot be used with the DMAC and the DTC.

- This module does not support transmission with 10-bit address.

- Multiple interrupts are not supported.

- API function calls except for the R_SCI_IIC_GetStatus function are disabled in the callback function.

- The I flag must be set to 1 to use interrupts.

- When using SCI (Simple I2C Mode) FIT Module and SCI Module Firmware Integration Technology (R01AN1815) in combination, the same channel cannot be used at the same time.

1.1 SCI Simple I2C Mode FIT Module

This module is implemented in a project and used as the API. Refer to 2.11 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_SCI_IIC_Open()	The function initializes the SCI simple I2C mode FIT module. This function must be called before calling any other API functions.
R_SCI_IIC_MasterSend()	Starts master transmission. Changes the transmit pattern according to the parameters. Operates batched processing until stop condition generation.
R_SCI_IIC_MasterReceive()	Starts master reception. Changes the receive pattern according to the parameters. Operates batched processing until stop condition generation.
R_SCI_IIC_Close()	This function completes the simple I2C communication and releases the SCI used.
R_SCI_IIC_GetStatus()	Returns the state of this module.
R_SCI_IIC_Control()	This function outputs conditions, Hi-Z from the SSDA pin, and one-shot of the SSCL clock. Also it resets the settings of this module. This function is mainly used when a communication error occurs.
R_SCI_IIC_GetVersion()	Returns the current version of this module.

1.3 Overview of SCI Simple I²C Mode FIT Module

1.3.1 Specifications of SCI Simple I²C Mode FIT Module

1. This module supports master transmission and 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.

2. An interrupt occurs when any of the following operations completes: start condition generation, slave address transmission, data reception, or stop condition generation. In the SCI (simple I²C mode) interrupt handling, the communication control function is called and the operation is continued.

3. The module supports multiple channels. When the device used has multiple channels, simultaneous communication is available using multiple channels.

4. Multiple slave devices on the same 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.

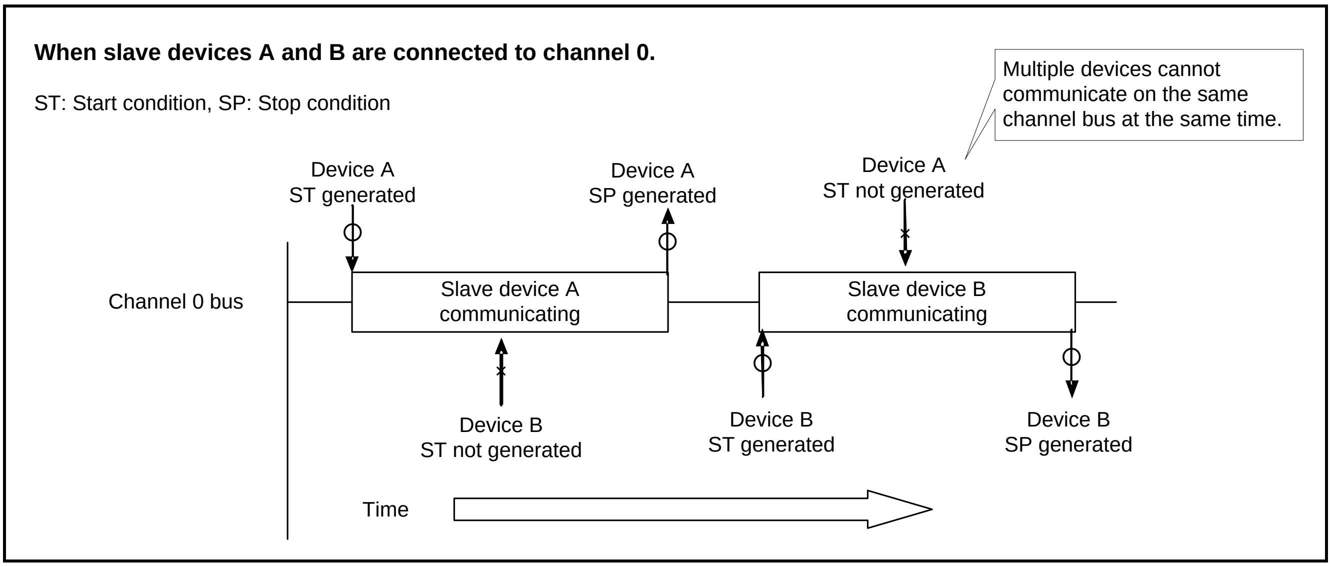


Figure 1.1 Example of Controlling Multiple Slave Devices

1.3.2 Master Transmission

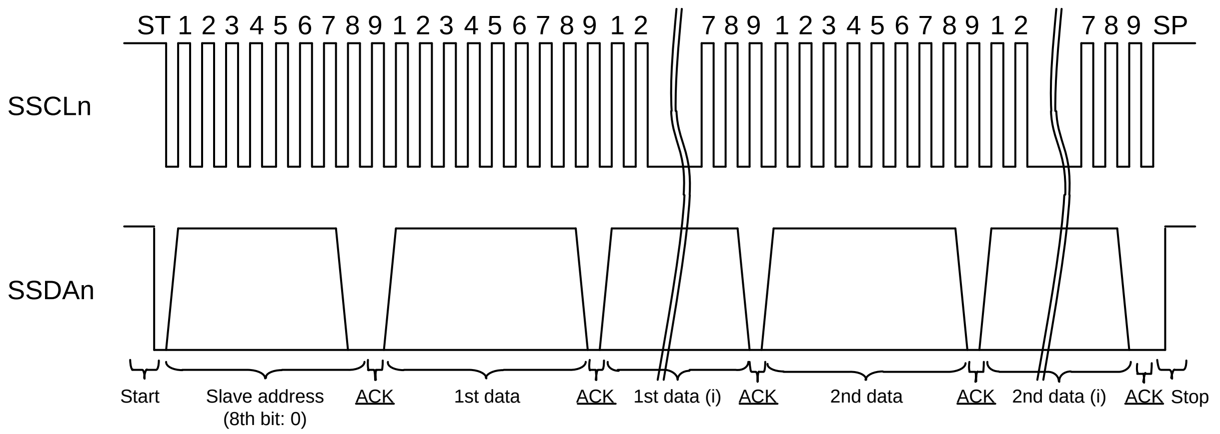
Data is transmitted from the master device (master (RX MCU)) to the slave device (slave).

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. Refer to 2.9 Parameters for details on the I²C communication information structure. Figure 1.2 to Figure 1.5 show the transmit patterns.

(1) Pattern 1

Data is transmitted from the master (RX MCU) 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

Data is transmitted from the master (RX MCU) to the slave. However, when the first data is not set, transmission for the first data is not performed.

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.

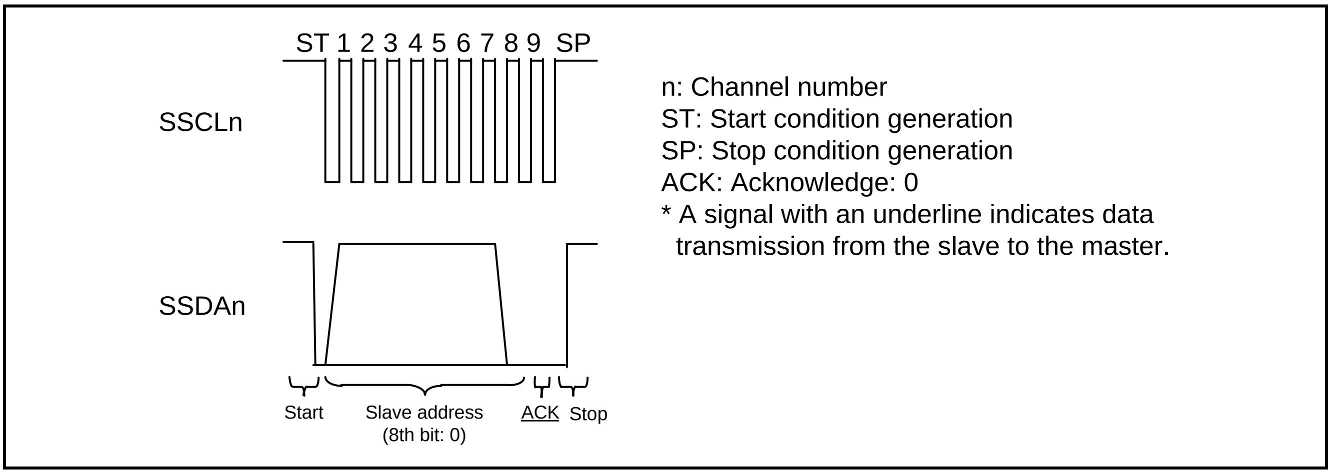


Figure 1.3 Signals for Pattern 2 of Master Transmission

(3) Pattern 3

Operations from start condition generation through to slave address transmission are the same as the operations for pattern 1. When 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.

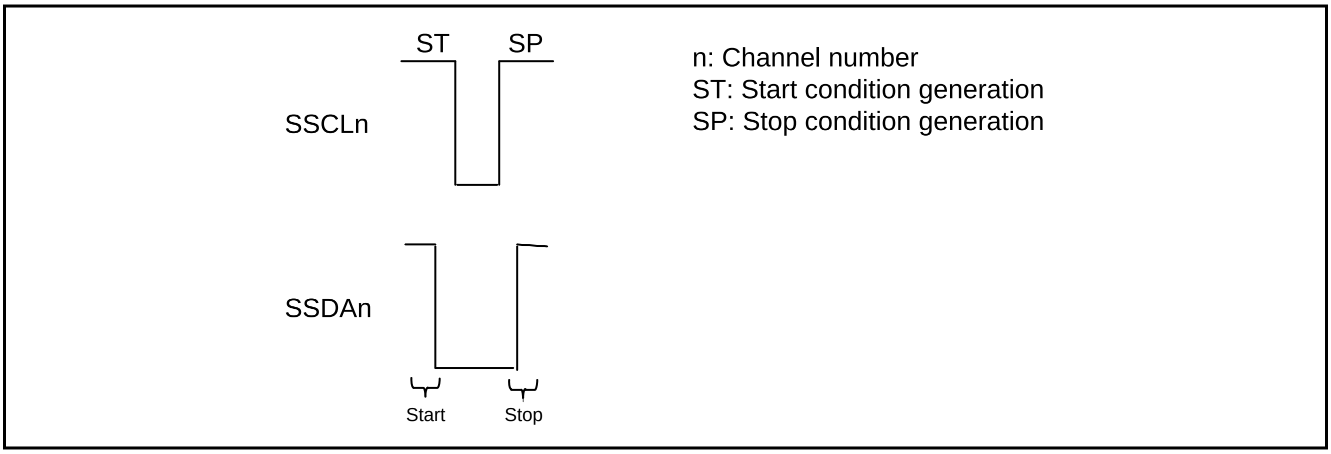


Figure 1.4 Signals for Pattern 3 of Master Transmission

(4) Pattern 4

After a start condition is generated, when the slave address, first data, and second data are not set, slave address transmission and data transmission are not performed. Then a stop condition is generated and the bus is released.

This pattern is useful for just releasing the bus.

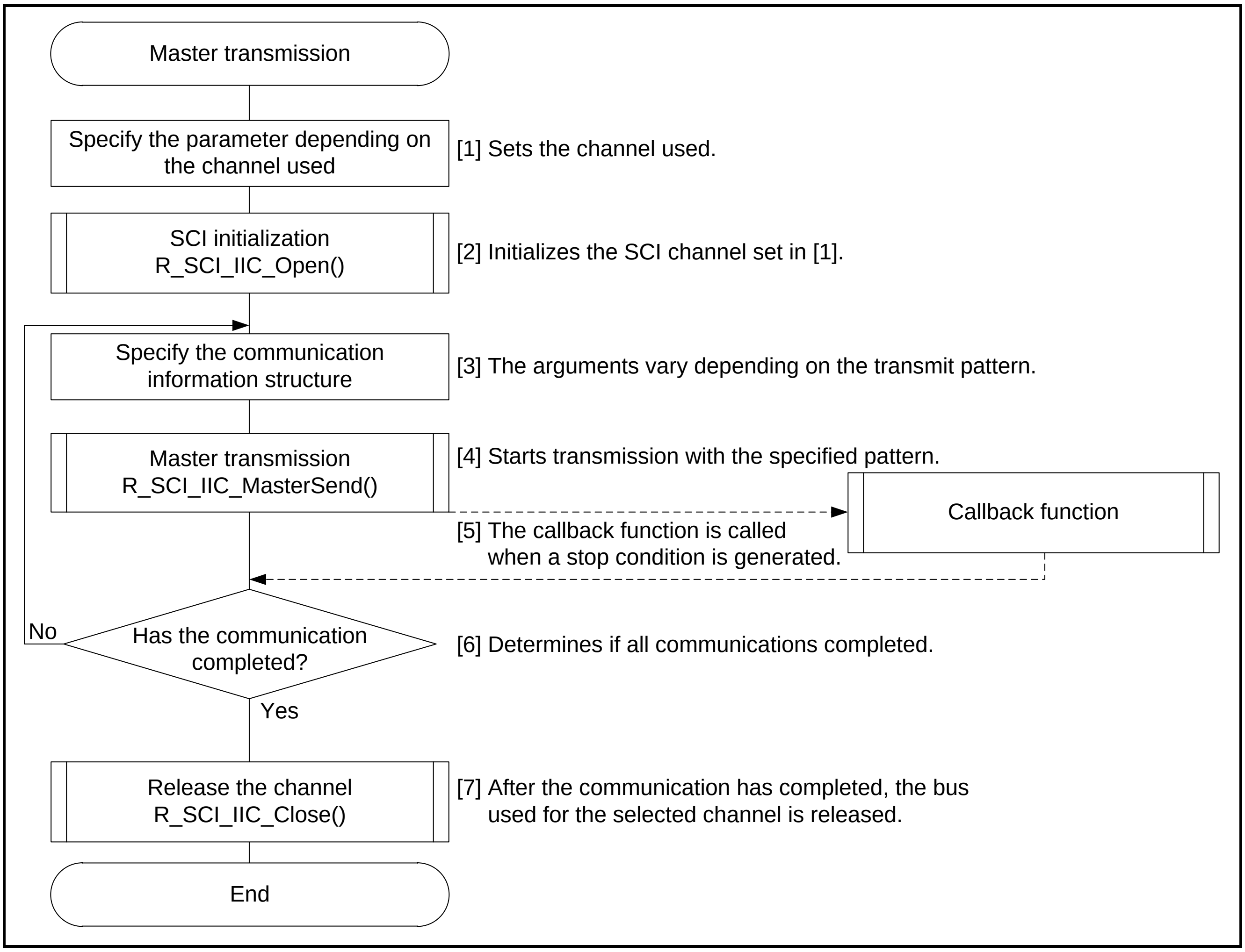
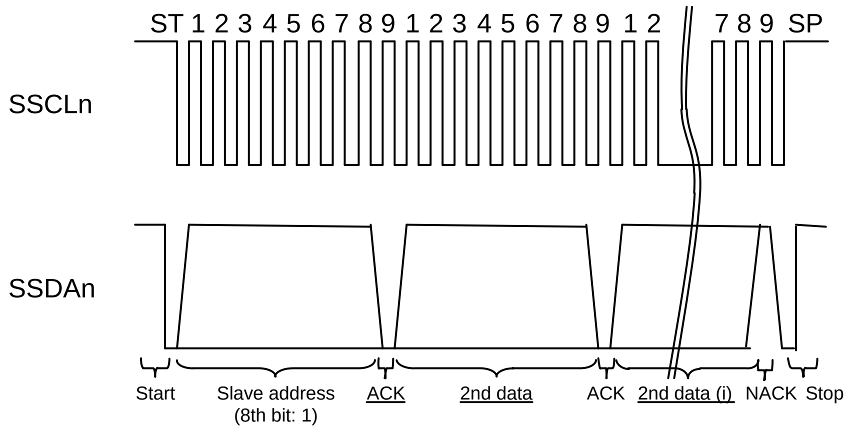


Figure 1.5 Signals for Pattern 4 of Master Transmission

Figure 1.6 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.



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

Figure 1.6 Example of Master Transmission

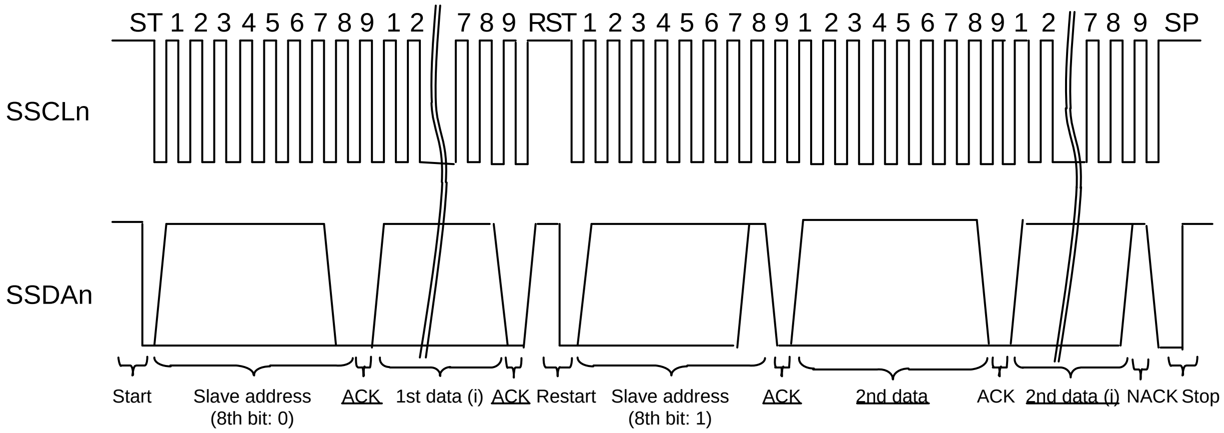
1.3.3 Master Reception

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. Refer to 2.9 Parameters for details on the I²C communication information structure. Figure 1.7 and Figure 1.8 show receive patterns.

(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.



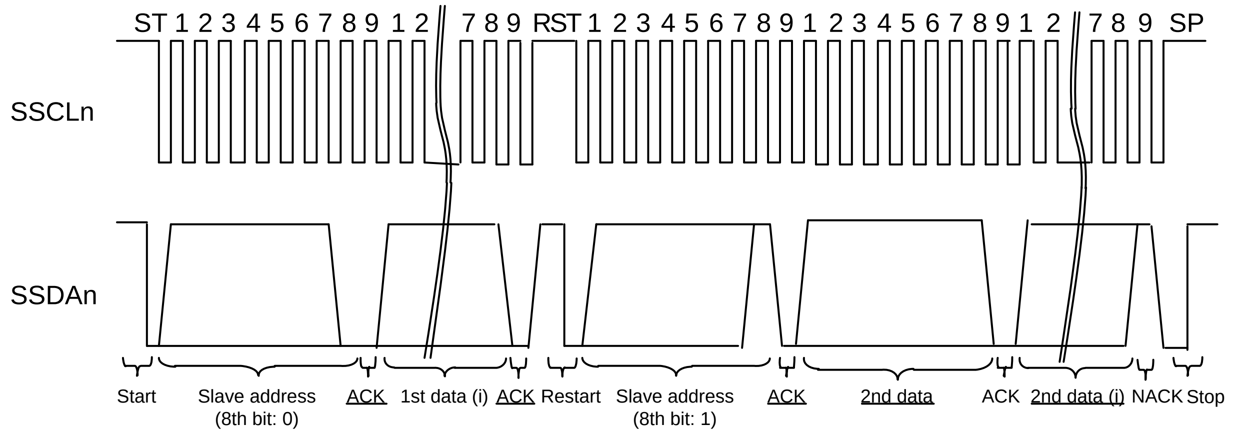
n: Channel number
ST: Start condition generation
SP: Stop condition generation
RST: Restart condition generation
NACK: Acknowledge: 1
ACK: Acknowledge: 0
* A signal with an underline indicates data transmission from the slave to the master.

Figure 1.7 Signals for Master Reception

(2) Master Transmit/Receive

The master (RX MCU) transmits data to the slave (master transmission). After the transmission completes, a restart condition is generated, the transfer direction is changed to 1 (read), and the master receives data from the slave (master reception).

A start condition is generated and then the slave address is transmitted. The eighth bit is the 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 NACK: Acknowledge: 1
SP: Stop condition generation ACK: Acknowledge: 0
RST: Restart condition generation
* A signal with an underline indicates data transmission from the slave to the master.

Figure 1.8 Signals for Master Transmit/Receive

Figure 1.9 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 I^{2}C communication information structure member.

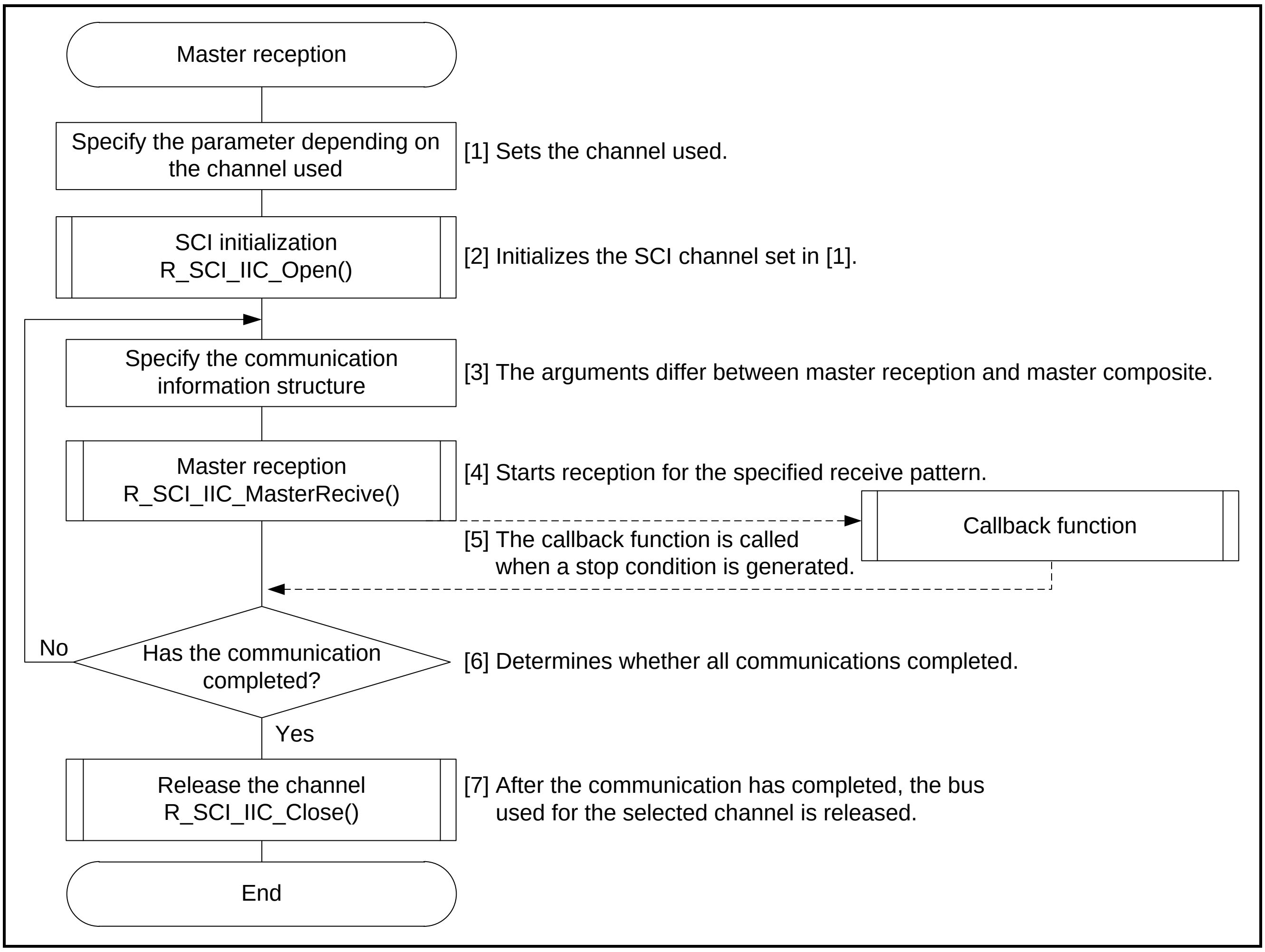


Figure 1.9 Example of Master Reception

1.3.4 State Transition

States entered in this module are uninitialized state, idle state, and communicating state.

Figure 1.10 shows the State Transition Diagram.

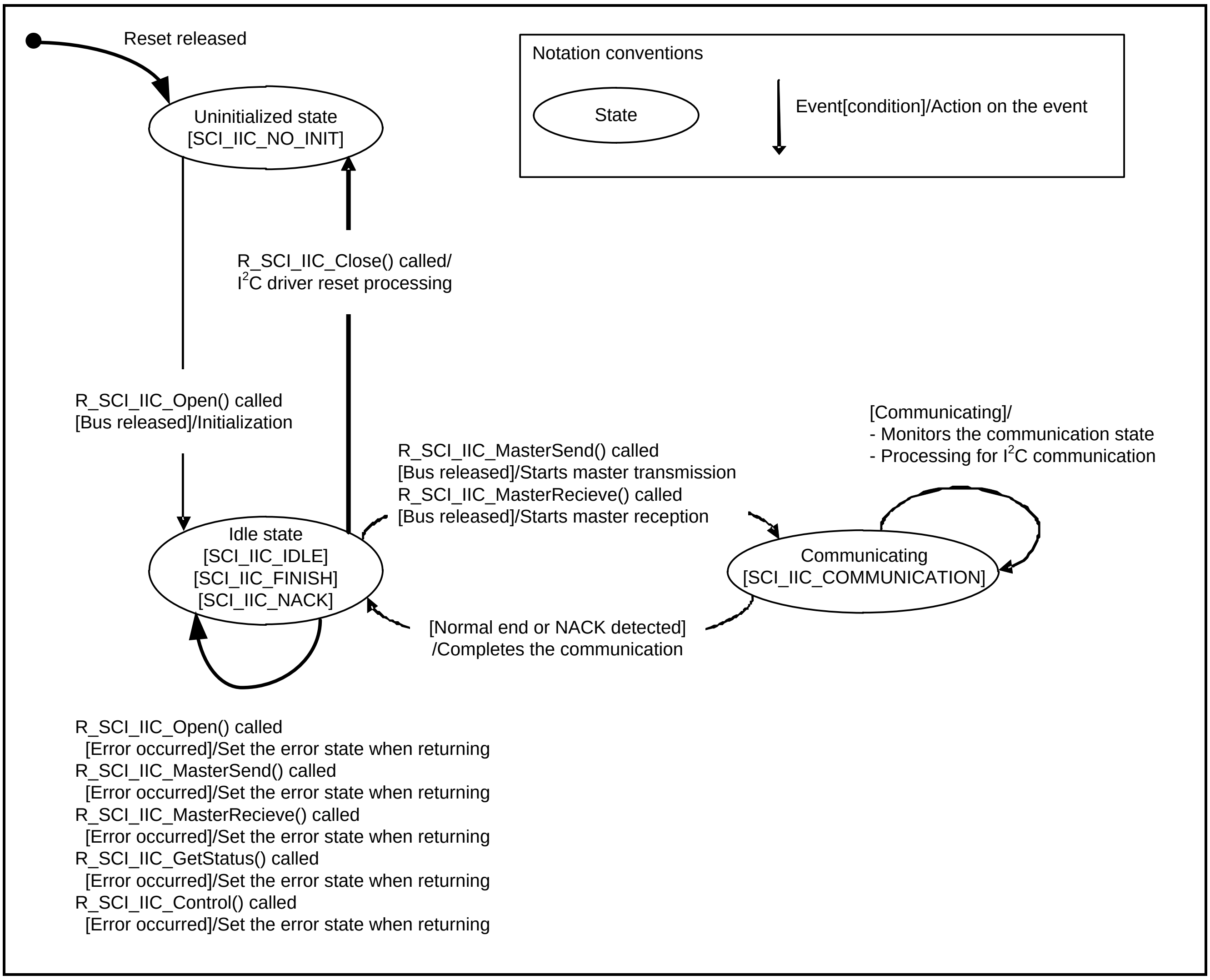


Figure 1.10 State Transition Diagram

1.3.5 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 same 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	SCI_IIC_NO_INIT
Idle states	SCI_IIC_IDLE SCI_IIC_FINISH SCI_IIC_NACK
Communicating (master transmission)	SCI_IIC_COMMUNICATION
Communicating (master reception)	SCI_IIC_COMMUNICATION
Communicating (master transmit/receive)	SCI_IIC_COMMUNICATION
Error	SCI_IIC_ERROR

2. API Information

This driver API adheres to the Renesas API naming standards.

2.1 Hardware Requirements

This driver requires your MCU supports the following feature:

- SCI

2.2 Software Requirements

This driver is dependent upon the following packages:

- Board Support Package Module (r_bsp)

2.3 Supported Toolchains

This driver is tested and works with the following toolchain:

- Renesas RX Toolchain v.2.02.00
- Renesas RX Toolchain v.2.03.00
- Renesas RX Toolchain v.2.05.00
- Renesas RX Toolchain v.2.06.00
- Renesas RX Toolchain v.2.07.00

Refer to 5.3 Operating Test Environment for details.

2.4 Usage of Interrupt Vector

The TXI interrupt and TEI interrupt are enabled by execution of R_SCI_IIC_MasterSend function or R_SCI_IIC_MasterReceive function (with specified condition)(while the macro definition SCI_IIC_CFG_CHi_INCLUDE (i = 0 to 12) is 1).

Table 2.1 to Table 2.3 shows the interrupt vectors used by the Simple I²C FIT module.

Table 2.1 List of Usage of Interrupt Vectors - 1 -

Device	Contents
RX110 RX111	TXI1 interrupt [channel 1] (vector no.: 220) TEI1 interrupt [channel 1] (vector no.: 221) TXI5 interrupt [channel 5] (vector no.: 224) TEI5 interrupt [channel 5] (vector no.: 225) TXI12 interrupt [channel 12] (vector no.: 240) TEI12 interrupt [channel 12] (vector no.: 241)
RX113 RX130 RX230 RX231	TXI0 interrupt [channel 0] (vector no.: 216) TEI0 interrupt [channel 0] (vector no.: 217) TXI1 interrupt [channel 1] (vector no.: 220) TEI1 interrupt [channel 1] (vector no.: 221) TXI5 interrupt [channel 5] (vector no.: 224) TEI5 interrupt [channel 5] (vector no.: 225) TXI6 interrupt [channel 6] (vector no.: 228) TEI6 interrupt [channel 6] (vector no.: 229) TXI8 interrupt [channel 8] (vector no.: 232) TEI8 interrupt [channel 8] (vector no.: 233) TXI9 interrupt [channel 9] (vector no.: 236) TEI9 interrupt [channel 9] (vector no.: 237) TXI12 interrupt [channel 12] (vector no.: 240) TEI12 interrupt [channel 12] (vector no.: 241)
RX23T	TXI1 interrupt [channel 1] (vector no.: 220) TEI1 interrupt [channel 1] (vector no.: 221) TXI5 interrupt [channel 5] (vector no.: 224) TEI5 interrupt [channel 5] (vector no.: 225)
RX24T	TXI1 interrupt [channel 1] (vector no.: 220) TEI1 interrupt [channel 1] (vector no.: 221) TXI5 interrupt [channel 5] (vector no.: 224) TEI5 interrupt [channel 5] (vector no.: 225) TXI6 interrupt [channel 6] (vector no.: 228) TEI6 interrupt [channel 6] (vector no.: 229)
RX24U	TXI1 interrupt [channel 1] (vector no.: 220) TEI1 interrupt [channel 1] (vector no.: 221) TXI5 interrupt [channel 5] (vector no.: 224) TEI5 interrupt [channel 5] (vector no.: 225) TXI6 interrupt [channel 6] (vector no.: 228) TEI6 interrupt [channel 6] (vector no.: 229) TXI8 interrupt [channel 8] (vector no.: 232) TEI8 interrupt [channel 8] (vector no.: 233) TXI9 interrupt [channel 9] (vector no.: 236) TEI9 interrupt [channel 9] (vector no.: 237) TXI11 interrupt [channel 11] (vector no.: 252) TEI11 interrupt [channel 11] (vector no.: 253)

Table 2.2 List of Usage of Interrupt Vectors - 2 -

Device	Contents
RX63N	TXI0 interrupt [channel 0] (vector no.: 215) TEI0 interrupt [channel 0] (vector no.: 216) TXI1 interrupt [channel 1] (vector no.: 218) TEI1 interrupt [channel 1] (vector no.: 219) TXI2 interrupt [channel 2] (vector no.: 221) TEI2 interrupt [channel 2] (vector no.: 222) TXI3 interrupt [channel 3] (vector no.: 224) TEI3 interrupt [channel 3] (vector no.: 225) TXI4 interrupt [channel 4] (vector no.: 227) TEI4 interrupt [channel 4] (vector no.: 228) TXI5 interrupt [channel 5] (vector no.: 230) TEI5 interrupt [channel 5] (vector no.: 231) TXI6 interrupt [channel 6] (vector no.: 233) TEI6 interrupt [channel 6] (vector no.: 234) TXI7 interrupt [channel 7] (vector no.: 236) TEI7 interrupt [channel 7] (vector no.: 237) TXI8 interrupt [channel 8] (vector no.: 239) TEI8 interrupt [channel 8] (vector no.: 240) TXI9 interrupt [channel 9] (vector no.: 242) TEI9 interrupt [channel 9] (vector no.: 243) TXI10 interrupt [channel 10] (vector no.: 245) TEI10 interrupt [channel 10] (vector no.: 246) TXI11 interrupt [channel 11] (vector no.: 248) TEI11 interrupt [channel 11] (vector no.: 249) TXI12 interrupt [channel 12] (vector no.: 250) TEI12 interrupt [channel 12] (vector no.: 252)
RX64M RX71M	TXI0 interrupt [channel 0] (vector no.: 59) TXI1 interrupt [channel 1] (vector no.: 61) TXI2 interrupt [channel 2] (vector no.: 63) TXI3 interrupt [channel 3] (vector no.: 81) TXI4 interrupt [channel 4] (vector no.: 83) TXI5 interrupt [channel 5] (vector no.: 85) TXI6 interrupt [channel 6] (vector no.: 87) TXI7 interrupt [channel 7] (vector no.: 99) TXI12 interrupt [channel 12] (vector no.: 117) GROUPBL0 interrupt (vector no.: 110) • TEI0 interrupt [channel 0] (group interrupt source no.: 0) • TEI1 interrupt [channel 1] (group interrupt source no.: 2) • TEI2 interrupt [channel 2] (group interrupt source no.: 4) • TEI3 interrupt [channel 3] (group interrupt source no.: 6) • TEI4 interrupt [channel 4] (group interrupt source no.: 8) • TEI5 interrupt [channel 5] (group interrupt source no.: 10) • TEI6 interrupt [channel 6] (group interrupt source no.: 12) • TEI7 interrupt [channel 7] (group interrupt source no.: 14) • TEI12 interrupt [channel 12] (group interrupt source no.: 16)

Table 2.3 List of Usage of Interrupt Vectors - 3 -

Device	Contents
RX65N RX651	TXI0 interrupt [channel 0] (vector no.: 59) TXI1 interrupt [channel 1] (vector no.: 61) TXI2 interrupt [channel 2] (vector no.: 63) TXI3 interrupt [channel 3] (vector no.: 81) TXI4 interrupt [channel 4] (vector no.: 83) TXI5 interrupt [channel 5] (vector no.: 85) TXI6 interrupt [channel 6] (vector no.: 87) TXI7 interrupt [channel 7] (vector no.: 99) TXI8 interrupt [channel 8] (vector no.: 101) TXI9 interrupt [channel 9] (vector no.: 103) TXI10 interrupt [channel 10] (vector no.: 105) TXI11 interrupt [channel 11] (vector no.: 115) TXI12 interrupt [channel 12] (vector no.: 117) GROUPBL0 interrupt (vector no.: 110) • TEI0 interrupt [channel 0] (group interrupt source no.: 0) • TEI1 interrupt [channel 1] (group interrupt source no.: 2) • TEI2 interrupt [channel 2] (group interrupt source no.: 4) • TEI3 interrupt [channel 3] (group interrupt source no.: 6) • TEI4 interrupt [channel 4] (group interrupt source no.: 8) • TEI5 interrupt [channel 5] (group interrupt source no.: 10) • TEI6 interrupt [channel 6] (group interrupt source no.: 12) • TEI7 interrupt [channel 7] (group interrupt source no.: 14) • TEI12 interrupt [channel 12] (group interrupt source no.: 16) GROUPBL1 interrupt (vector no.: 111) • TEI8 interrupt [channel 8] (group interrupt source no.: 24) • TEI9 interrupt [channel 9] (group interrupt source no.: 26) GROUPAL0 interrupt (vector no.: 112) • TEI10 interrupt [channel 10] (group interrupt source no.: 8) • TEI11 interrupt [channel 11] (group interrupt source no.: 12)

2.5 Header Files

All API calls and their supporting interface definitions are located in r_sci_iic_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_sci_iic_rx_config.h and r_sci_iic_rx_pin_config.h. The option names and setting values are listed in the table below.

Configuration options in *r_sci_iic_rx_config.h* (1/2)	
`SCI_IIC_CFG_PARAM_CHECKING_ENABLE` **`- Default value = 1`**	Selectable whether to include parameter checking in the code. - When this is set to 0, parameter checking is omitted. - When this is set to 1, parameter checking is included.
`SCI_IIC_CFG_CHi_INCLUDED` **`i = 0 to 12`** **`- When i = 0 to 12, the default value = 0`**	Selectable whether to use available channels. - 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. To use a channel, please change the definition value of the channel to be used to 1.
`SCI_IIC_CFG_CHi_BITRATE_BPS` **`i = 0 to 12`** **`- Default value = 384000 for all`**	Specifies the bit rate. Specify a value less than or equal to 384000 (384 kbit/sec.). The bit rate setting should be based on this definition value and the clock setting definition value specified by RX Family Board Support Package Module (BSP FIT module). Depending on the target device to be used and the BSP FIT module clock setting, the actual bit rate may differ from the expected bit rate.
`SCI_IIC_CFG_CHi_INT_PRIORITY` **`i = 0 to 12`** **`- Default value = 2 for all`**	Specifies interrupt priority levels for condition generation, receive-data-full, transmit-data-empty, and transmit-end interrupts. Specify the level between 1 and 15.
`SCI_IIC_CFG_CHi_DIGITAL_FILTER` **`i = 0 to 12`** **`- Default value = 1 for all`**	Selectable whether to use the noise cancellation function for the SSCL and SSDA input signals. - When this is set to 0, the noise cancellation function is disabled. - When this is set to 1, the noise cancellation function is enabled.
`SCI_IIC_CFG_CHi_FILTER_CLOCK` **`i = 0 to 12`** **`- Default value = 1 for all`**	Select the sampling clock used for digital noise filter. - When this is set to 1, the clock divided by 1 is used. - When this is set to 2, the clock divided by 2 is used. - When this is set to 3, the clock divided by 4 is used. - When this is set to 4, the clock divided by 8 is used.
`SCI_IIC_CFG_CHi_SSDA_DELAY_SELECT` **`i = 0 to 12`** **`- Default value = 18 for all`**	Select the delay time for output on the SSDA pin relative to the falling edge of the output on the SSCL pin. Specify the delay between 1 and 31. The default value is a value based on PCLK which operates in 60 MHz and is the clock source of the on-chip baud rate generator. The SSDA delay time is increased or decreased according to the clock source of the on-chip baud rate generator. When the bit rate or the PCLK frequency is set to low speed, the SSDA falling timing may occur after the SSCL falling timing in the start condition. Confirm and set an appropriate value depending on the user system.

Configuration options in *r_sci_iic_rx_config.h* (2/2)	
SCI_IIC_CFG_BUS_CHECK_COUNTER **i = 0 to 12** **- Default value = 1000**	Specifies the timeout counter (number of times to perform bus checking) when the simple I^{2}C API function performs bus checking. Specify a value less than or equal to 0xFFFFFFFF. The bus checking is performed after generating each condition using the simple I^{2}C control function (R_SCI_IIC_Control function). With the bus checking, the timeout counter is decremented after generating each condition. When the counter reaches 0, the API determines that a timeout has occurred and returns an error (Busy) as the return value. * The timeout counter is used for the bus not to be locked by the bus lock or others. Therefore specify the value greater than or equal to the time for that the other device holds the SCL pin low. Setting time for the timeout (ns) $\approx (\frac{1}{\text{ICLK}}$ (Hz)) $\times$ counter value $\times$ 10
SCI_IIC_CFG_PORT_SETTING_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_SCI_IIC_CFG_SCIi_SSCLi_PORT, R_SCI_IIC_CFG_SCIi_SSCLi_BIT, R_SCI_IIC_CFG_SCIi_SSDAi_PORT, and R_SCI_IIC_CFG_SCIi_SSDAi_BIT as pins SSCL and SSDA. - 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. - When you assume this setting 0, please set four definitions mentioned above.

Configuration options in *r_sci_iic_rx_pin_config.h* (1/2)	

`R_SCI_IIC_CFG_SCIi_SSCLi_PORT`
`i = 0 to 12`
`- When i = 0, the default value = '2'`
`- When i = 1, the default value = '1'`
`- When i = 2, the default value = '5'`
`- When i = 3, the default value = '2'`
`- When i = 4, the default value = 'B'`
`- When i = 5, the default value = 'B'`
`- When i = 6, the default value = 'B'`
`- When i = 7, the default value = '9'`
`- When i = 8, the default value = 'C'`
`- When i = 9, the default value = 'B'`
`- When i = 10, the default value = '8'`
`- When i = 11, the default value = '7'`
`- When i = 12, the default value = 'E'`
	Selects port groups used as the SSCL pins.
	Specify the value as an ASCII code in the range '0' to 'J'.

`R_SCI_IIC_CFG_SCIi_SSCLi_BIT`
`i = 0 to 12`
`- When i = 0, the default value = '1'`
`- When i = 1, the default value = '5'`
`- When i = 2, the default value = '2'`
`- When i = 3, the default value = '5'`
`- When i = 4, the default value = '0'`
`- When i = 5, the default value = '1'`
`- When i = 6, the default value = '1'`
`- When i = 7, the default value = '2'`
`- When i = 8, the default value = '6'`
`- When i = 9, the default value = '6'`
`- When i = 10, the default value = '1'`
`- When i = 11, the default value = '6'`
`- When i = 12, the default value = '2'`
	Selects pins used as the SSCL pins.
	Specify the value as an ASCII code in the range '0' to '7'.

`R_SCI_IIC_CFG_SCIi_SSDAi_PORT`
`i = 0 to 12`
`- When i = 0, the default value = '2'`
`- When i = 1, the default value = '1'`
`- When i = 2, the default value = '5'`
`- When i = 3, the default value = '2'`
`- When i = 4, the default value = 'B'`
`- When i = 5, the default value = 'B'`
`- When i = 6, the default value = 'B'`
`- When i = 7, the default value = '9'`
`- When i = 8, the default value = 'C'`
`- When i = 9, the default value = 'B'`
`- When i = 10, the default value = '8'`
`- When i = 11, the default value = '7'`
`- When i = 12, the default value = 'E'`
	Selects port groups used as the SSDA pin.
	Specify the value as an ASCII code in the range '0' to 'J'.

Configuration options in *r_sci_iic_rx_pin_config.h* (2/2)	

`R_SCI_IIC_CFG_SCIi_SSDAi_BIT`
`i = 0 to 12`
`- When i = 0, the default value = ‘0’`
`- When i = 1, the default value = ‘6’`
`- When i = 2, the default value = ‘0’`
`- When i = 3, the default value = ‘3’`
`- When i = 4, the default value = ‘1’`
`- When i = 5, the default value = ‘2’`
`- When i = 6, the default value = ‘2’`
`- When i = 7, the default value = ‘0’`
`- When i = 8, the default value = ‘7’`
`- When i = 9, the default value = ‘7’`
`- When i = 10, the default value = ‘2’`
`- When i = 11, the default value = ‘7’`
`- When i = 12, the default value = ‘1’`
	Selects port groups used as the SSDA pin.
	Specify the value as an ASCII code in the range ‘0’ to ‘7’.

2.8 Code Size

Typical code sizes associated with this module are listed below. Information is listed for a single representative device of the RX100 Series, RX200 Series, and RX600 Series, respectively.

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 the C compiler's compile options are set to their default values, as described in 2.3, Supported Toolchains. 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.

ROM, RAM and Stack Code Sizes					
Device	Category		Memory Used		Remarks
			With Parameter Checking	Without Parameter Checking	
RX130	ROM	1 channel used	4,306 bytes	4,189 bytes	
		2 channels used	4,454 bytes	4,337 bytes	The ROM size can be calculated using the following formula: 1-channel usage + (148 bytes × number of additional channels)
	RAM	1 channel used	41 bytes		
		2 channels used	69 bytes		The RAM size can be calculated using the following formula: 1-channel usage + (28 bytes × number of additional channels)
	Maximum stack usage		256 bytes		Nested interrupts are prohibited, so the maximum value when one channel is used is listed.
RX231	ROM	1 channel used	4,298 bytes	4,181 bytes	
		2 channels used	4,446 bytes	4,329 bytes	The ROM size can be calculated using the following formula: 1-channel usage + (148 bytes × number of additional channels)
	RAM	1 channel used	41 bytes		
		2 channels used	69 bytes		The RAM size can be calculated using the following formula: 1-channel usage + (28 bytes × number of additional channels)
	Maximum stack usage		224 bytes		Nested interrupts are prohibited, so the maximum value when one channel is used is listed.
RX64M	ROM	1 channel used	4,347 bytes	4,230 bytes	
		2 channels used	4,493 bytes	4,376 bytes	The ROM size can be calculated using the following formula: 1-channel usage + (146 bytes × number of additional channels)
	RAM	1 channel used	41 bytes		
		2 channels used	69 bytes		The RAM size can be calculated using the following formula: 1-channel usage + (28 bytes × number of additional channels)
	Maximum stack usage		224 bytes		Nested interrupts are prohibited, so the maximum value when one channel is used is listed.

2.9 Parameters

This section describes the structure whose members are API parameters. This structure is located in r_sci_iic_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 (SCI_IIC_COMMUNICATION).

```
typedef struct
{
    uint8_t rsv2; /* Reserved area */
    uint8_t rsv1; /* Reserved area */
    sci_iic_ch_dev_status_t dev_sts; /* Device state flag */
    uint8_t ch_no;  /* Channel number for the device used */
    sci_iic_callback callbackfunc;      /* Callback function */
    uint32_t cnt2nd;/* Second data counter (number of bytes) */
    uint32_t cnt1st;/* First data counter (number of bytes) */
    uint8_t * p_data2nd;   /* Pointer to the buffer to store the second data */
    uint8_t * p_data1st;   /* Pointer to the buffer to store the first data */
    uint8_t * p_slv_adr;  /* Pointer to the buffer to store the slave address */
} sci_iic_info_t;
```

2.10 Return Values

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

```
typedef enum  /* Simple I2C-bus API state codes */
{
    SCI_IIC_SUCCESS, /* Processing completed successfully */
    SCI_IIC_ERR_LOCK_FUNC, /* Multiple calls occurred on the same channel. */
    SCI_IIC_ERR_INVALID_CHAN, /* Nonexistent channel */
    SCI_IIC_ERR_INVALID_ARG, /* Invalid parameter */
    SCI_IIC_ERR_NO_INIT, /* Uninitialized state */
    SCI_IIC_ERR_BUS_BUSY, /* Bus is busy. This state occurs with the following cases: */
                           /* The initialization function or a start function is */
                            /* called during communication. */
                           /* A start function or advance function is called while */
                            /* another device on the same channel is communicating. */
    SCI_IIC_ERR_OTHER /* Other error */
} sci_iic_return_t;
```

2.11 Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends using “Smart Configurator” described in (1) or (3). However, “Smart Configurator” only supports some RX devices. Please use the methods of (2) or (4) for unsupported RX devices.

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

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

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

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

3. API Functions

3.1 R_SCI_IIC_Open()

The function initializes the simple I²C FIT module. This function must be called before calling any other API functions.

Format

```
sci_iic_return_t R_SCI_IIC_Open(
        sci_iic_info_t * p_sci_iic_info  /* Structure data */
)
```

Parameters

** p_sci_iic_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 (SCI_IIC_COMMUNICATION).

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

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

Return Values

SCI_IIC_SUCCESS / Processing completed successfully */*
SCI_IIC_ERR_LOCK_FUNC / The API is locked by the other task. */*
SCI_IIC_ERR_INVALID_CHAN / Nonexistent channel */*
SCI_IIC_ERR_INVALID_ARG / Invalid parameter */*
SCI_IIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_sci_iic_rx_if.h.

Description

Performs the initialization to start the simple I²C-bus communication. Sets the SCI channel specified by the parameter. If the state of the channel is 'uninitialized (SCI_IIC_NO_INIT)', the following processes are performed.

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

The bit rate set in initial setting to start simple I²C-bus communication.

The bit rate is set based on the setting value of "2.7 Configuration Overview" and the clock setting definition value specified by BSP FIT module.

Reentrant

Function is reentrant for different channels.

Example

```
volatile sci_iic_return_t ret;
sci_iic_info_t            siic_info;

siic_info.dev_sts = SCI_IIC_NO_INIT;
siic_info.ch_no = 1;

ret = R_SCI_IIC_Open(&siic_info);
```

Special Notes

None

3.2 R_SCI_IIC_MasterSend()

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

Format

```
sci_iic_return_t R_SCI_IIC_MasterSend(
        sci_iic_info_t * p_sci_iic_info  /* Structure data */
)
```

Parameters

** p_sci_iic_info*

This is the pointer to the I2C communication information structure. The transmit patterns can be selected from four patterns by the parameter. Refer to the 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 (SCI_IIC_COMMUNICATION).

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.

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

Return Values

SCI_IIC_SUCCESS / Processing completed successfully */*
SCI_IIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
SCI_IIC_ERR_INVALID_ARG / The parameter is invalid. */*
SCI_IIC _ERR_NO_INIT / Uninitialized state */*
SCI_IIC _ERR_BUS_BUSY / The bus state is busy. */*
SCI_IIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_sci_iic_rx_if.h.

Description

Starts the simple I2C-bus master transmission. The transmission is performed with the SCI channel and transmit pattern specified by parameters. If the state of the channel is 'idle (SCI_IIC_IDEL)', the following processes are performed.

- Setting the state flag
- Initializing variables used by the API
- Enabling the SCI interrupts
- Releasing the I2C reset
- Allocating I2C output ports
- Generating a start condition

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

- Either SCL or SDA line is in low state.

The transmission processing is performed sequentially in subsequent interrupt processing after this function return SCI_IIC_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 "5.2.1 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_sci_iic_ChStatus [], that is to be "SCI_IIC_FINISH" for normal completion.

Notes:

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

Reentrant

Function is reentrant for different channels.

Example

```
- Case1: Transmit pattern 1

#include <stddef.h>     // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

void main(void);
void Callback_ch1(void);

void main(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_info_t            siic_info;

    uint8_t slave_addr_eeprom[1] = {0x50};  /* Slave address for EEPROM */
    uint8_t access_addr1[1]      = {0x00};  /* 1st data field */
    uint8_t send_data[5]         = {0x81,0x82,0x83,0x84,0x85};

    /* Sets IIC Information (Send pattern 1) */
    siic_info.p_slv_adr    = slave_addr_eeprom;
```

```
    siic_info.p_data1st    = access_addr1;
    siic_info.p_data2nd    = send_data;
    siic_info.dev_sts      = SCI_IIC_NO_INIT;
    siic_info.cnt1st       = 1;
    siic_info.cnt2nd       = 3;
    siic_info.callbackfunc = &Callback_ch1;
    siic_info.ch_no        = 1;

    /* SCI open */
    ret = R_SCI_IIC_Open(&siic_info);
    /* Start Master Send */
    ret = R_SCI_IIC_MasterSend(&siic_info);

    if (SCI_IIC_SUCCESS == ret)
    {
        while(SCI_IIC_FINISH != siic_info.dev_sts);
    }
    else
    {
        /* error */
    }

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

void Callback_ch1(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t      iic_status;
    sci_iic_info_t            iic_info_ch;

    iic_info_ch.ch_no = 1;
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* Call error processing for the R_SCI_IIC_GetStatus()function*/
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
        /* Processing when a NACK is detected
           by verifying the iic_status flag. */
        }
    }
}
```

- Case2: Transmitting data to two slave devices (Slave 1 and slave 2) continuously.

```
#include <stddef.h>     // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

void main(void);
void Callback_ch1(void);

void main(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_info_t            siic_info_slave1;
    sci_iic_info_t            siic_info_slave2;

    uint8_t slave_addr_eeprom[1]  = {0x50}; /* Slave address for EEPROM */
    uint8_t slave_addr_m16c[1]    = {0x01}; /* Slave address for M16C */
    uint8_t write_addr_slave1[1]  = {0x01};     /* 1st data field */
    uint8_t write_addr_slave2[1]  = {0x02};     /* 1st data field */
    uint8_t data_area_slave1[5]   = {0x81,0x82,0x83,0x84,0x85};
    uint8_t data_area_slave2[5]   = {0x18,0x28,0x38,0x48,0x58};

    /* Sets ‘Slave 1’ Information (Send pattern 1) */
    siic_info_slave1.p_slv_adr = slave_addr_eeprom;
    siic_info_slave1.p_data1st = write_addr_slave1;
    siic_info_slave1.p_data2nd = data_area_slave1;
    siic_info_slave1.dev_sts = SCI_IIC_NO_INIT;
    siic_info_slave1.cnt1st = 1;
    siic_info_slave1.cnt2nd = 3;
    siic_info_slave1.callbackfunc = &Callback_ch1;
    siic_info_slave1.ch_no = 1;

    /* SCI open */
    ret = R_SCI_IIC_Open(&siic_info_slave1);
    /* Start Master Send */
    ret = R_SCI_IIC_MasterSend(&siic_info_slave1);

    while((SCI_IIC_FINISH != siic_info_slave1.dev_sts) &&
          (SCI_IIC_NACK != siic_info_slave1.dev_sts));

    /* Sets ‘Slave 2’ Information (Send pattern 1) */
    siic_info_slave2.p_slv_adr = slave_addr_m16c;
    siic_info_slave2.p_data1st = write_addr_slave2;
    siic_info_slave2.p_data2nd = data_area_slave2;
    siic_info_slave2.dev_sts = SCI_IIC_NO_INIT;
    siic_info_slave2.cnt1st = 1;
    siic_info_slave2.cnt2nd = 3;
    siic_info_slave2.callbackfunc = &Callback_ch1;
    siic_info_slave2.ch_no = 1;

    /* Start Master Send */
    ret = R_SCI_IIC_MasterSend(&siic_info_slave2);

    while((SCI_IIC_FINISH != siic_info_slave2.dev_sts) &&
          (SCI_IIC_NACK != siic_info_slave2.dev_sts));
    while(1);
}
```

To access multiple slave devices, rewrite the information structure for each slave device to be accessed.

```
void Callback_ch1(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t      iic_status;
    sci_iic_info_t            iic_info_ch;

    iic_info_ch.ch_no = 1;
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* Call error processing for the R_SCI_IIC_GetStatus()function*/
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
        /* Processing when a NACK is detected
           by verifying the iic_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	Pattern 4
*p_slv_adr	Buffer pointer to the slave address storage			FIT_NO_PTR [1]
*p_data1st	Buffer pointer to the first data storage	FIT_NO_PTR [1]	FIT_NO_PTR [1]	FIT_NO_PTR [1]
*p_data2nd	Buffer pointer to the second data (transmit data) storage		FIT_NO_PTR [1]	FIT_NO_PTR [1]
dev_sts	Device state flag			
cnt1st	0000 0001h to FFFF FFFFh [2]	0	0	0
cnt2nd	0000 0001h to FFFF FFFFh [2]		0	0
callbackfunc	Specify the function name used			
ch_no	00h to FFh			
rsv1, rsv2, rsv3	Reserved (value set here has no effect)			

Notes:

1. When using pattern 2, 3, or 4, set ‘FIT_NO_PTR’ as the argument of the parameter.
2. Do not set to 0.

3.3 R_SCI_IIC_MasterReceive()

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

Format

sci_iic_return_t R_SCI_IIC_MasterRecive(

sci_iic_info_t * p_sci_iic_info /* Structure data */

)

Parameters

** p_sci_iic_info*

This is the pointer to the I²C communication information structure. The receive pattern can be selected from master reception and master transmit/receive. 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 (SCI_IIC_COMMUNICATION).

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.

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

Return Values

SCI_IIC_SUCCESS / Processing completed successfully */*
SCI_IIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
SCI_IIC_ERR_INVALID_ARG / The parameter is invalid. */*
SCI_IIC _ERR_NO_INIT / Uninitialized state */*
SCI_IIC _ERR_BUS_BUSY / The bus state is busy. */*
SCI_IIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_sci_iic_rx_if.h.

Description

Starts the simple I2C-bus master reception. The reception is performed with the SCI channel and receive pattern specified by parameters. If the state of the channel is ‘idle (SCI_IIC_IDEL)’, the following processes are performed.

- Setting the state flag
- Initializing variables used by the API
- Enabling the SCI interrupts
- Releasing the I2C reset
- Allocating I2C output ports
- Generating a start condition

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

- Either SCL or SDA line is in low state.

The reception processing is performed sequentially in subsequent interrupt processing after this function return SCI_IIC_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 "5.2.2 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_sci_iic_ChStatus [], that is to be "SCI_IIC_FINISH" for normal completion.

Notes:

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

Reentrant

Function is reentrant for different channels.

Example

```
#include <stddef.h>     // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

void main(void);
void Callback_ch1(void);

void main(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_info_t            siic_info;

    uint8_t slave_addr_eeprom[1] = {0x50}; /* Slave address for EEPROM */
    uint8_t access_addr1[1]      = {0x00}; /* 1st data field           */
    uint8_t store_area[5]        = {0xFF,0xFF,0xFF,0xFF,0xFF};

    /* Sets IIC Information (Ch1) */
    siic_info.p_slv_adr = slave_addr_eeprom;
    siic_info.p_data1st = access_addr1;
```

```
    siic_info.p_data2nd = store_area;
    siic_info.dev_sts = SCI_IIC_NO_INIT;
    siic_info.cnt1st = 1;
    siic_info.cnt2nd = 3;
    siic_info.callbackfunc = &Callback_ch1;
    siic_info.ch_no = 1;

    /* SCI open */
    ret = R_SCI_IIC_Open(&siic_info);
    /* Start Master Receive */
    ret = R_SCI_IIC_MasterReceive(&siic_info);
    if (SCI_IIC_SUCCESS == ret)
    {
        while(SCI_IIC_FINISH != siic_info.dev_sts);
    }
    else
    {
        /* error */
    }

    /* Master receive complete */
    while(1);
}

void Callback_ch1(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t      iic_status;
    sci_iic_info_t            iic_info_ch;

    iic_info_ch.ch_no = 1;
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* Call error processing for the R_SCI_IIC_GetStatus()function*/
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
        /* Processing when a NACK is detected
           by verifying the iic_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	Buffer pointer to the slave address storage	
*p_data1st	(Value set here has no effect)	Buffer pointer to the first data storage
*p_data2nd	Buffer pointer to the second data (receive data) storage	
dev_sts	Device state flag	
cnt1st (1)	0	0000 0001h to FFFF FFFFh
cnt2nd (2)	0000 0001h to FFFF FFFFh	0000 0001h to FFFF FFFFh
callbackfunc	Specify the function name used	
ch_no	00h to FFh	
rsv1, rsv2, rsv3	Reserved (value set here has no effect)	

Notes:

1. The receive pattern is determined by whether cnt1st is 0 or not.
2. Do not set to 0.

3.4 R_SCI_IIC_Close()

This function completes the simple I2C communication and releases the SCI used.

Format

```
sci_iic_return_t R_SCI_IIC_Close(
        sci_iic_info_t * p_sci_iic_info /* Structure data */
)
```

Parameters

** p_sci_iic_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 (SCI_IIC_COMMUNICATION).

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

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

Return Values

SCI_IIC_SUCCESS / Processing completed successfully */*
SCI_IIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
SCI_IIC_ERR_INVALID_ARG / The parameter is invalid. */*

Properties

Prototyped in r_sci_iic_rx_if.h.

Description

Configures the settings to complete the simple I2C-bus communication. Disables the SCI channel specified by the parameter. The following processes are performed in this function.

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

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

Reentrant

Function is reentrant for different channels.

Example

```
volatile sci_iic_return_t ret;
sci_iic_info_t            siic_info;

siic_info.ch_no = 1;

ret = R_SCI_IIC_Close(&siic_info);
```

Special Notes

None

3.5 R_SCI_IIC_GetStatus()

Returns the state of this module.

Format

```
sci_iic_return_t R_SCI_IIC_GetStatus(
        sci_iic_info_t * p_sci_iic_info /* Structure data */
        sci_iic_mcu_status_t *p_sci_iic_status /* State of this module */
)
```

Parameters

** p_sci_iic_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 (SCI_IIC_COMMUNICATION).

```
uint8_t ch_no;  /* Channel number */
```

**p_sci_iic_status*

This contains the address to store the I²C state flag. If the argument is 'FIT_NO_PTR', the state is not returned.
Use the structure members listed below to specify parameters.

```
typedef union
{
    uint32_t                LONG;
    struct st_sci_iic_status_flag
{
    uint32_t rsv :27 /* Reserve bit */
    uint32_t SCLI:1; /* SSCL pin level */
    uint32_t SDAI:1; /* SSDA pin level */
    uint32_t NACK :1; /* NACK detection flag */
    uint32_t TRS :1; /* Transmit/receive mode level */
    uint32_t BSY :1; /* Bus state flag */
    }BIT;
} sci_iic_mcu_status_t;
```

Return Values

SCI_IIC_SUCCESS / Processing completed successfully */*
SCI_IIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
SCI_IIC_ERR_INVALID_ARG / The parameter is invalid. */*
SCI_IIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_sci_iic_rx_if.h.

Description

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

Reentrant

Function is reentrant for different channels.

Example

```
volatile sci_iic_return_t ret;
sci_iic_info_t           siic_info;
sci_iic_mcu_status_t     iic_status;

siic_info.ch_no = 1

ret = R_SCI_IIC_GetStatus(&siic_info, &iic_status);
```

Special Notes

The following shows the state flag allocation.

b31 to b16
Reserved
Reserved
rsv
Always 0

b15 to b8
Reserved
Reserved
rsv
Always 0

b7 to b5	b4	b3	b2	b1	b0
Reserved	Pin level		Event detection	Mode	Bus state
Reserved	SSCL pin level	SSDA pin level	NACK detection	Send/ receive mode	Bus busy/ready
rsv	SCLI	SDAI	NACK	TRS	BSY
Always 0	0: Low level 1: High level		0: Not detected 1: Detected	0: Receive 1: Transmit	0: Idle 1: Busy

3.6 R_SCI_IIC_Control()

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

Format

```
sci_iic_return_t R_SCI_IIC_Control(
        r_sci_iic_info_t * p_sci_iic_info /* Structure data */
        sci_iic_ctrl_ptn_t ctrl_ptn /* Output pattern */
);
```

Parameters

** p_sci_iic_info*

This is the pointer to the I^{2}C 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 (SCI_IIC_COMMUNICATION).

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

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

ctrl_ptn_t

Specifies the output pattern. When selecting multiple options, specify them with '|'.

<u>The following options can be selected simultaneously:</u>

- The following three options can be specified simultaneously. Then they will be processed in the order listed.
  - SCI_IIC_GEN_START_CON
  - SCI_IIC_GEN_RESTART_CON
  - SCI_IIC_GEN_STOP_CON
- The following two options can be specified simultaneously.
  - SCI_IIC_GEN_SDA_HI_Z
  - SCI_IIC_GEN_SSCL_ONESHOT

```
typedef uint8_t sci_iic_ctrl_ptn_t;
#define SCI_IIC_GEN_START_CON (sci_iic_ctrl_ptn_t)(0x01)
                                     /* Start condition generation */
#define SCI_IIC_GEN_STOP_CON (sci_iic_ctrl_ptn_t)(0x02)
                                     /* Stop condition generation */
#define SCI_IIC_GEN_RESTART_CON (sci_iic_ctrl_ptn_t)(0x04)
                                     /* Restart condition generation */
#define SCI_IIC_GEN_SSDA_HI_Z (sci_iic_ctrl_ptn_t)(0x08)
                                     /* Hi-Z output from the SSDA pin */
#define SCI_IIC_GEN_SSCL_ONESHOT (sci_iic_ctrl_ptn_t)(0x10)
                                     /* SSCL clock one-shot output */
#define SCI_IIC_GEN_RESET (sci_iic_ctrl_ptn_t)(0x20)
                                     /* Simple I2C mode reset */
```

Return Values

SCI_IIC_SUCCESS / Processing completed successfully */*
SCI_IIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
SCI_IIC_ERR_INVALID_ARG / The parameter is invalid. */*
SCI_IIC _ERR_BUS_BUSY / The bus state is busy. */*
SCI_IIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_sci_iic_rx_if.h.

Description

Outputs control signals of the simple I²C mode. Outputs conditions specified by the argument, Hi-Z from the SSDA pin, and one-shot of the SSCL clock. Also resets the simple I²C mode settings.

Reentrant

Function is reentrant for different channels.

Example

```
volatile sci_iic_return_t ret;
sci_iic_info_t            siic_info;

siic_info.ch_no = 1;

/* Output an extra SSCL clock cycle after changes the SSDA pin in a high-
impedance state */
ret = R_SCI_IIC_Control(&siic_info, SCI_IIC_GEN_SSDA_HI_Z |
SCI_IIC_SSCL_ONESHOT);
```

Special Notes

None

3.7 R_SCI_IIC_GetVersion()

Returns the current version of this module.

Format

uint32_t R_SCI_IIC_GetVersion(void)

Parameters

None

Return Values

Version number

Properties

Prototyped in r_sci_iic_rx_if.h.

Description

This function will return the version of the currently installed SCI (simple I2C mode) 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.

Reentrant

Function is reentrant for different channels.

Example

```
uint32_t version;

version = R_SCI_IIC_GetVersion();
```

Special Notes

This function is inlined using ‘#pragma inline’.

4. Pin Settings

To use the SCI (Simple I^{2}C Mode) 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 SCI (Simple I^{2}C Mode) FIT module can choose whether or not to perform the pin setting in the R_SCI_IIC_Open / R_SCI_IIC_MasterSend / R_SCI_IIC_MasterReceive / R_SCI_IIC_Close / R_SCI_IIC_Control function depending on the setting of the configuration option SCI_IIC_CFG_PORT_SET_PROCESSING.

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

When performing the Pin Setting in the e^2 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_sci_iic_pin_config.h file. Values of the macro definitions listed in Table 4.1 and Table 4.2 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 SCI (Simple I^{2}C Mode) FIT module.

Table 4.1 Macro Definitions for the Pin Setting Feature – 1 –

Channel Selected	Pin Selected	Macro Definition
Channel 0	SSCL0 Pin	R_SCI_IIC_CFG_SCI0_SSCL0_PORT R_SCI_IIC_CFG_SCI0_SSCL0_BIT
	SSDA0 Pin	R_SCI_IIC_CFG_SCI0_SSDA0_PORT R_SCI_IIC_CFG_SCI0_SSDA0_BIT
Channel 1	SSCL1 Pin	R_SCI_IIC_CFG_SCI1_SSCL1_PORT R_SCI_IIC_CFG_SCI1_SSCL1_BIT
	SSDA1 Pin	R_SCI_IIC_CFG_SCI1_SSDA1_PORT R_SCI_IIC_CFG_SCI1_SSDA1_BIT
Channel 2	SSCL2 Pin	R_SCI_IIC_CFG_SCI2_SSCL2_PORT R_SCI_IIC_CFG_SCI2_SSCL2_BIT
	SSDA2 Pin	R_SCI_IIC_CFG_SCI2_SSDA2_PORT R_SCI_IIC_CFG_SCI2_SSDA2_BIT
Channel 3	SSCL3 Pin	R_SCI_IIC_CFG_SCI3_SSCL3_PORT R_SCI_IIC_CFG_SCI3_SSCL3_BIT
	SSDA3 Pin	R_SCI_IIC_CFG_SCI3_SSDA3_PORT R_SCI_IIC_CFG_SCI3_SSDA3_BIT
Channel 4	SSCL4 Pin	R_SCI_IIC_CFG_SCI4_SSCL4_PORT R_SCI_IIC_CFG_SCI4_SSCL4_BIT
	SSDA4 Pin	R_SCI_IIC_CFG_SCI4_SSDA4_PORT R_SCI_IIC_CFG_SCI4_SSDA4_BIT
Channel 5	SSCL5 Pin	R_SCI_IIC_CFG_SCI5_SSCL5_PORT R_SCI_IIC_CFG_SCI5_SSCL5_BIT
	SSDA5 Pin	R_SCI_IIC_CFG_SCI5_SSDA5_PORT R_SCI_IIC_CFG_SCI5_SSDA5_BIT
Channel 6	SSCL6 Pin	R_SCI_IIC_CFG_SCI6_SSCL6_PORT R_SCI_IIC_CFG_SCI6_SSCL6_BIT
	SSDA6 Pin	R_SCI_IIC_CFG_SCI6_SSDA6_PORT R_SCI_IIC_CFG_SCI6_SSDA6_BIT

Table 4.2 Macro Definitions for the Pin Setting Feature – 2 –

Channel Selected	Pin Selected	Macro Definition
Channel 7	SSCL7 Pin	R_SCI_IIC_CFG_SCI7_SSCL7_PORT R_SCI_IIC_CFG_SCI7_SSCL7_BIT
	SSDA7 Pin	R_SCI_IIC_CFG_SCI7_SSDA7_PORT R_SCI_IIC_CFG_SCI7_SSDA7_BIT
Channel 8	SSCL8 Pin	R_SCI_IIC_CFG_SCI8_SSCL8_PORT R_SCI_IIC_CFG_SCI8_SSCL8_BIT
	SSDA8 Pin	R_SCI_IIC_CFG_SCI8_SSDA8_PORT R_SCI_IIC_CFG_SCI8_SSDA8_BIT
Channel 9	SSCL9 Pin	R_SCI_IIC_CFG_SCI9_SSCL9_PORT R_SCI_IIC_CFG_SCI9_SSCL9_BIT
	SSDA9 Pin	R_SCI_IIC_CFG_SCI9_SSDA9_PORT R_SCI_IIC_CFG_SCI9_SSDA9_BIT
Channel 10	SSCL10 Pin	R_SCI_IIC_CFG_SCI10_SSCL10_PORT R_SCI_IIC_CFG_SCI10_SSCL10_BIT
	SSDA10 Pin	R_SCI_IIC_CFG_SCI10_SSDA10_PORT R_SCI_IIC_CFG_SCI10_SSDA10_BIT
Channel 11	SSCL11 Pin	R_SCI_IIC_CFG_SCI11_SSCL11_PORT R_SCI_IIC_CFG_SCI11_SSCL11_BIT
	SSDA11 Pin	R_SCI_IIC_CFG_SCI11_SSDA11_PORT R_SCI_IIC_CFG_SCI11_SSDA11_BIT
Channel 12	SSCL12 Pin	R_SCI_IIC_CFG_SCI12_SSCL12_PORT R_SCI_IIC_CFG_SCI12_SSCL12_BIT
	SSDA12 Pin	R_SCI_IIC_CFG_SCI12_SSDA12_PORT R_SCI_IIC_CFG_SCI12_SSDA12_BIT

Pins selected in the r_sci_iic_pin_config.h file are configured as peripheral function pins SSCL and SSDA after calling the R_SCI_IIC_MasterSend / R_SCI_IIC_MasterReceive / R_SCI_IIC_Control function.

The pins assigned to the peripheral function are released when the communication operation executed by the R_SCI_IIC_MasterSend / R_SCI_IIC_MasterReceive / R_SCI_IIC_Control function is completed or upon calling the R_SCI_IIC_Close function and then become general I/O pins (as input pins).

Pins SSCL and SSDA must be pulled up with an external resistor.

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

5. Appendices

5.1 Communication Method

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

5.1.1 States for API Operation

Table 5.1 lists the States Used for Protocol Control.

Table 5.1 States Used for Protocol Control (enum sci_iic_api_status_t)

No.	Constant Name	Description
STS0	SCI_IIC_STS_NO_INIT	Uninitialized state
STS1	SCI_IIC_STS_IDLE	Idle state
STS2	SCI_IIC_STS_ST_COND_WAIT	Wait state for a start condition to be generated
STS3	SCI_IIC_STS_SEND_SLVADR_W_WAIT	Wait state for the slave address [write] transmission to complete
STS4	SCI_IIC_STS_SEND_SLVADR_R_WAIT	Wait state for the slave address [read] transmission to complete
STS5	SCI_IIC_STS_SEND_DATA_WAIT	Wait state for the data transmission to complete
STS6	SCI_IIC_STS_RECEIVE_DATA_WAIT	Wait state for the data reception to complete
STS7	SCI_IIC_STS_SP_COND_WAIT	Wait state for a stop condition to be generated

5.1.2 Events During API Operation

Table 5.2 lists the Events Used for Protocol Control. When the interface functions accompanying this module are called, they are defined as events as well as interrupts.

Table 5.2 Events Used for Protocol Control (enum sci_iic_api_event_t)

No.	Event	Event Definition
EV0	SCI_IIC_EV_INIT	sci_iic_init_driver() called
EV1	SCI_IIC_EV_GEN_START_COND	sci_iic_generate_start_cond() called
EV2	SCI_IIC_EV_INT_START	STI interrupt occurred (interrupt flag: START)
EV3	SCI_IIC_EV_INT_ADD	TXI interrupt occurred
EV4	SCI_IIC_EV_INT_SEND	TXI interrupt occurred
EV5	SCI_IIC_EV_INT_STOP	STI interrupt occurred (interrupt flag: STOP)
EV6	SCI_IIC_EV_INT_NACK	STI interrupt occurred (interrupt flag: NACK)

5.1.3 Protocol State Transitions

In this module, a state transition occurs when an interface function provided is called or when an SCI (simple I^{2}C mode) interrupt request is generated. Figure 5.1 to Figure 5.4 show protocol state transitions.

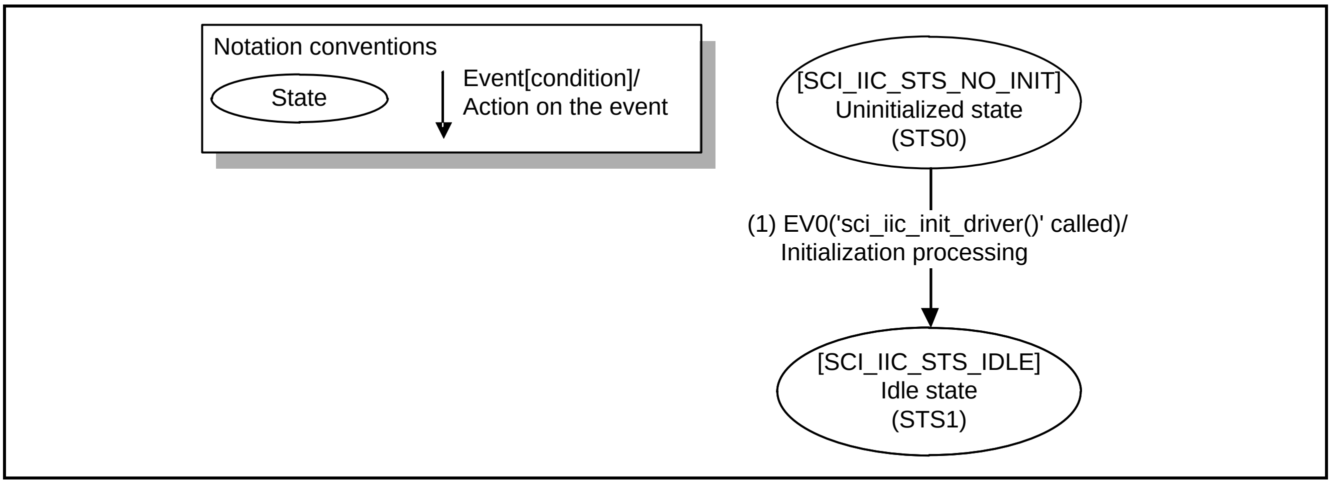


Figure 5.1 State Transition on Initialization

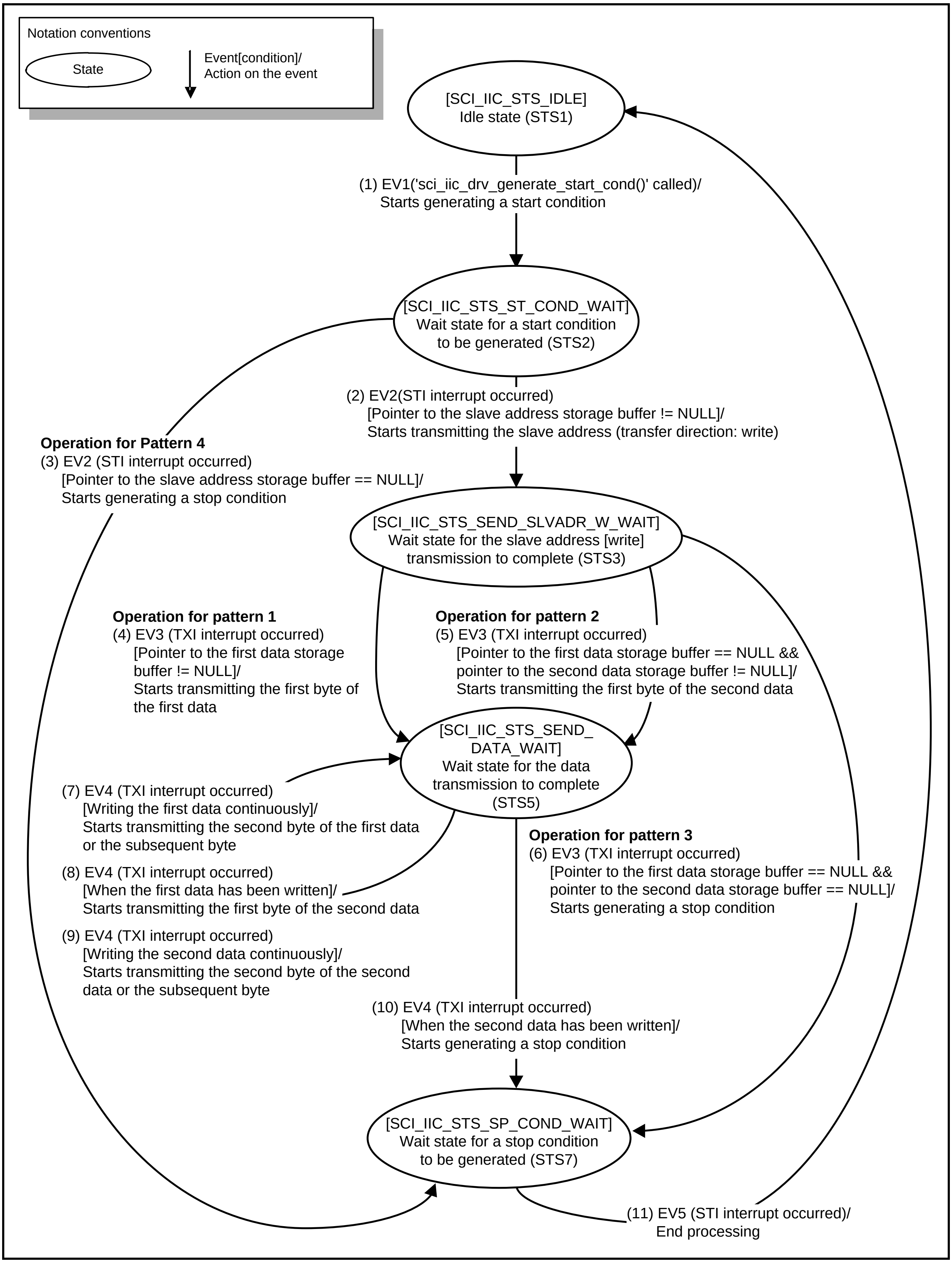


Figure 5.2 State Transition on Master Transmission

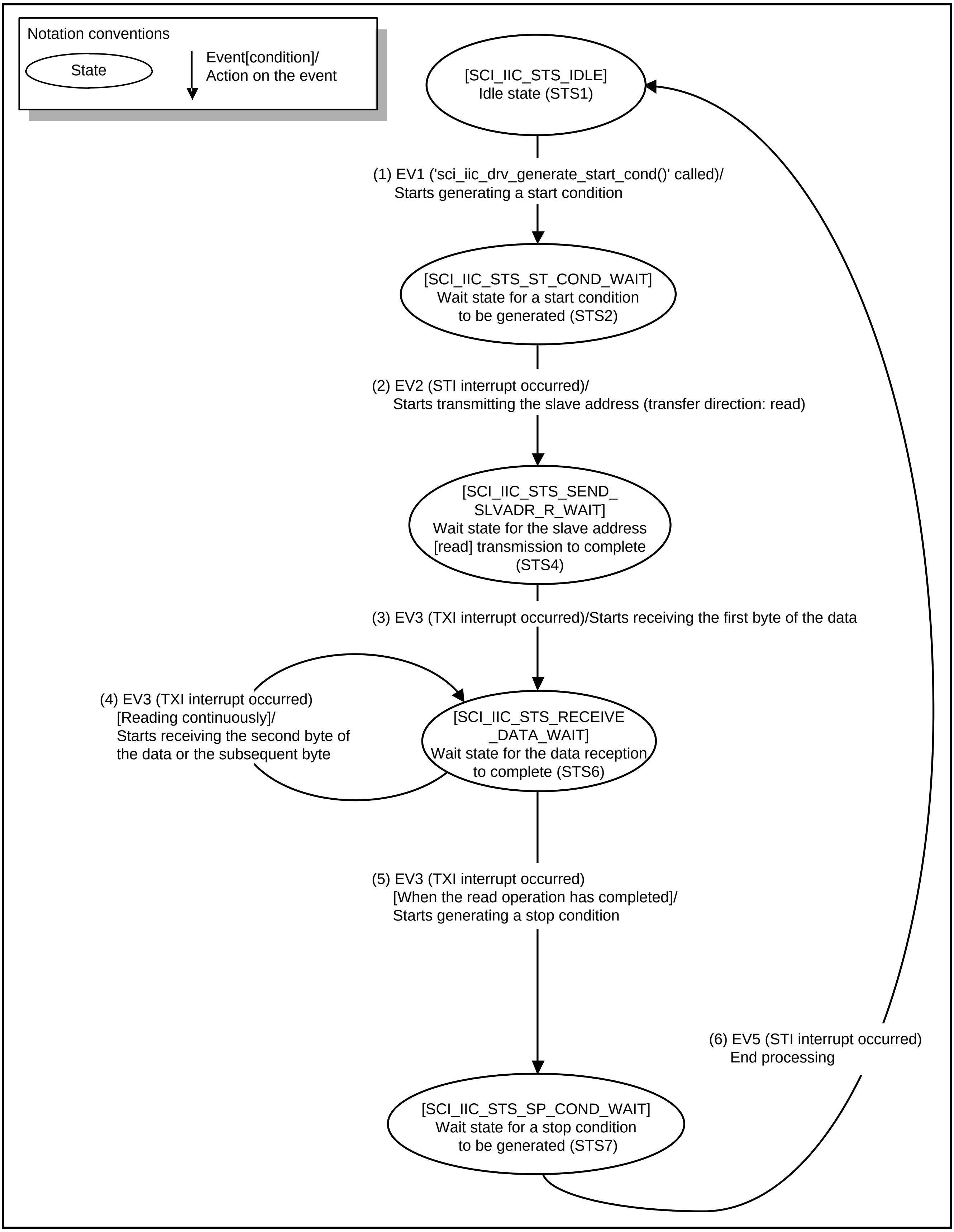


Figure 5.3 State Transition on Master Reception

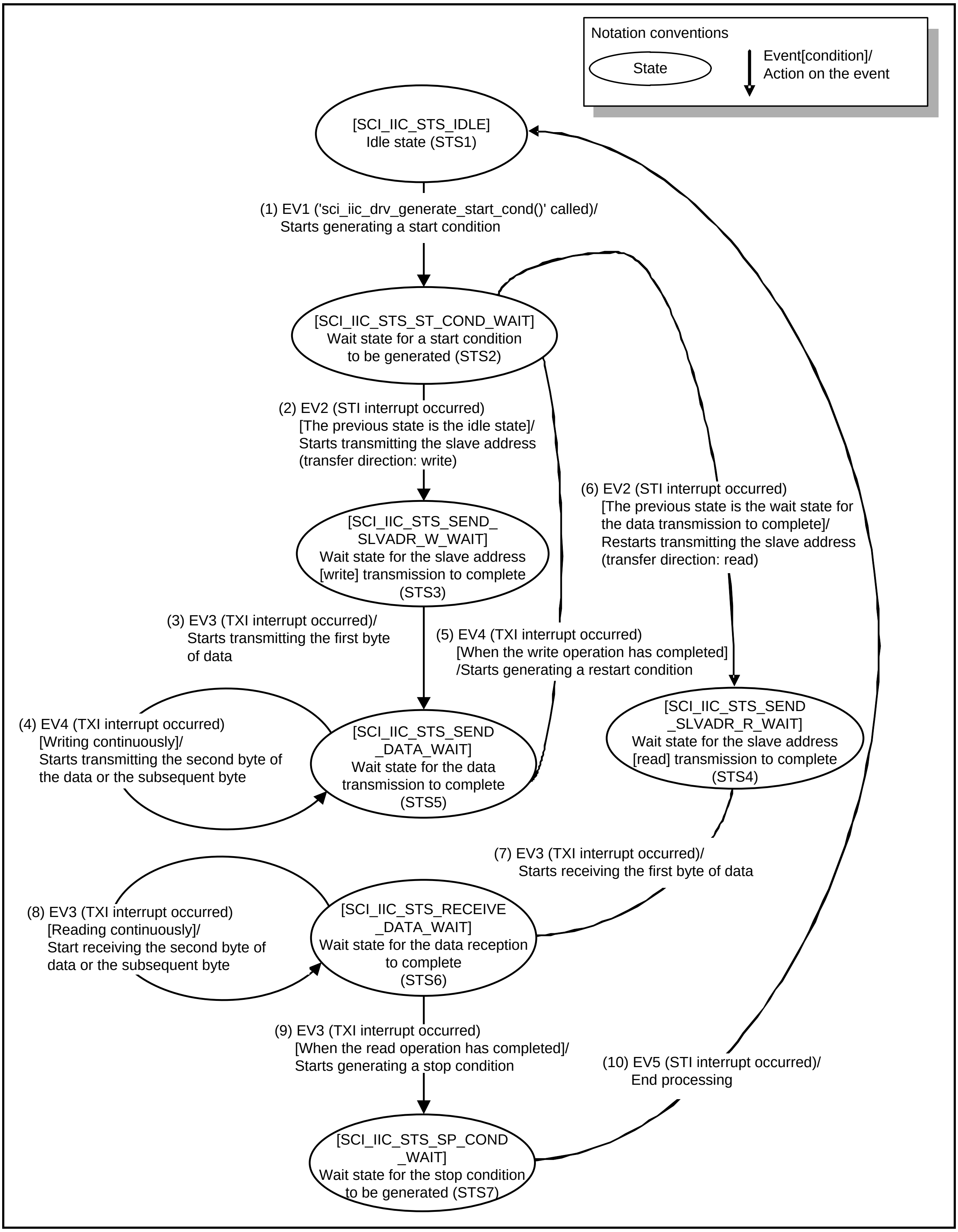


Figure 5.4 State Transition on Master Transmit/Receive

5.1.4 Protocol State Transition Table

The processing when the events in Table 5.2 occur in the states in Table 5.1 is shown in the Table 5.3 Protocol State Transition. Refer to Table 5.4 for details of each function.

Table 5.3 Protocol State Transition Table (gc_sci_iic_mtx_tbl[][]) (1)

State		Event						
		EV0	EV1	EV2	EV3	EV4	EV5	EV6
STS0	Uninitialized state [SCI_IIC_STS_NO_INIT]	Func0	ERR	ERR	ERR	ERR	ERR	ERR
STS1	Idle state [SCI_IIC_STS_IDLE]	ERR	Func1	ERR	ERR	ERR	ERR	ERR
STS2	Wait state for a start condition to be generated SCI_IIC_STS_ST_COND_WAIT	ERR	ERR	Func2	ERR	ERR	ERR	Func7
STS3	Wait state for the slave address [write] transmission to complete [SCI_IIC_STS_SEND_SLVADR_W_WAIT]	ERR	ERR	ERR	Func3	ERR	ERR	Func7
STS4	Wait state for the slave address [read] transmission to complete [SCI_IIC_STS_SEND_SLVADR_R_WAIT]	ERR	ERR	ERR	Func3	ERR	ERR	Func7
STS5	Wait state for the data transmission to complete [SCI_IIC_STS_SEND_DATA_WAIT]	ERR	ERR	ERR	ERR	Func4	ERR	Func7
STS6	Wait state for the data reception to complete [SCI_IIC_STS_RECEIVE_DATA_WAIT]	ERR	ERR	ERR	Func5	ERR	ERR	Func7
STS7	Wait state for the stop condition to be generated [SCI_IIC_STS_SP_COND_WAIT]	ERR	ERR	ERR	ERR	ERR	Func6	Func7

Note:

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

5.1.5 Functions Used on Protocol State Transitions

Table 5.4 lists the Functions Used on Protocol State Transition.

Table 5.4 Functions Used on Protocol State Transition

Processing	Function	Overview
Func0	sci_iic_init_driver()	Initialization
Func1	sci_iic_generate_start_cond()	Start condition generation
Func2	sci_iic_after_gen_start_cond()	Processing after generating a start condition
Func3	sci_iic_after_send_slvadr()	Processing after transmitting the slave address
Func4	sci_iic_write_data_sending()	Data transmission
Func5	sci_iic_read_data_receiving()	Data reception
Func6	sci_iic_release()	Communication end processing
Func7	sci_iic_nack()	NACK error processing

5.1.6 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_sci_iic_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 'SCI_IIC_IDLE/SCI_IIC_FINISH/SCI_IIC_NACK' (idle state) and communication is available. When the bus is being used for communication, the flag becomes 'SCI_IIC_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 5.5 lists States of Flags on State Transitions.

Table 5.5 States of Flags on State Transitions

State	Channel State Flag g_sci_iic_ChStatus[]	Device State Flag (Communication Device) I2C Communication Information Structure *p_dev_sts	I2C Protocol Operating Mode Internal Communication Information Structure api_Mode	Current State of the Protocol Control Internal Communication Information Structure api_N_status
Uninitialized state	SCI_IIC_NO_INIT	SCI_IIC_NO_INIT	SCI_IIC_MODE_NONE	SCI_IIC_STS_NO_INIT
Idle state	SCI_IIC_IDLE SCI_IIC_FINISH SCI_IIC_NACK	SCI_IIC_IDLE SCI_IIC_FINISH SCI_IIC_NACK	SCI_IIC_MODE_NONE	SCI_IIC_STS_IDLE
Communicating (master transmission)	SCI_IIC_ COMMUNICATION	SCI_IIC_ COMMUNICATION	SCI_IIC_MODE_WRITE	SCI_IIC_STS_ST_COND_WAIT
				SCI _IIC_STS_SEND_SLVADR_W_WAIT
				SCI _IIC_STS_SEND_DATA_WAIT
				SCI _IIC_STS_SP_COND_WAIT
Communicating (master reception)	SCI_IIC_ COMMUNICATION	SCI_IIC_ COMMUNICATION	SCI_IIC_MODE_READ	SCI_IIC_STS_ST_COND_WAIT
				SCI_IIC_STS_SEND_SLVADR_R_WAIT
				SCI_IIC_STS_RECEIVE_DATA_WAIT
				SCI_IIC_STS_SP_COND_WAIT
Communicating (master transmit/receive)	SCI_IIC_ COMMUNICATION	SCI_IIC_ COMMUNICATION	SCI_IIC_MODE_ COMBINED	SCI_IIC_STS_ST_COND_WAIT
				SCI_IIC_STS_SEND_SLVADR_W_WAIT
				SCI_IIC_STS_SEND_SLVADR_R_WAIT
				SCI_IIC_STS_SEND_DATA_WAIT
				SCI_IIC_STS_RECEIVE_DATA_WAIT
				SCI_IIC_STS_SP_COND_WAIT
Error state	SCI_IIC_ERROR	SCI_IIC_ERROR	—	—

5.2 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

5.2.1 Master Transmission

(1) Pattern 1

ST	AD6 to AD0	/W	/ACK	D7 to D0	/ACK	D7 to D0	/ACK	SP	
▲1			▲2		▲3		▲4	▲5	

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

▲2: TXI interrupt: Address transmission completed (transfer direction bit: write) (1)

▲3: TXI interrupt: Data transmission completed (first data) (1)

▲4: TXI interrupt: Data transmission completed (second data) (1)

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

(2) Pattern 2

ST	AD6 to AD0	/W	/ACK	D7 to D0	/ACK	SP	
▲1			▲2		▲3	▲4	

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

▲2: TXI interrupt: Address transmission completed (transfer direction bit: write) (1)

▲3: TXI interrupt: Data transmission completed (second data) (1)

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

Note:

1. An interrupt request is generated on the rising edge of the ninth clock.

(3) Pattern 3

ST	AD6 to AD0	/W	/ACK	SP

▲1 ▲2 ▲3

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

▲2: TXI interrupt: Address transmission completed (transfer direction bit: write) (1)

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

(4) Pattern 4

ST	SP

▲1 ▲2

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

▲2: STI (STOP) interrupt: Stop condition detected

Note:

1. An interrupt request is generated on the rising edge of the ninth clock.

5.2.2 Master Reception

ST	AD6 to AD0	R	/ACK	D7 to D0	/ACK	D7 to D0	NACK	SP

▲1 ▲2 ▲3 ▲4 ▲5

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

▲2: TXI interrupt: Address transmission completed (transfer direction bit: read) (1)

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

▲4: TXI interrupt: Reception for the last data completed (second data) (2)

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

Notes:

1. An interrupt request is generated on the rising edge of the ninth clock.
2. An interrupt request is generated on the rising edge of the eighth clock.

5.2.3 Master Transmit/Receive

ST	AD6 to AD0	/W	/ACK	D7 to D0	/ACK	RST	AD6 to AD0	R
	▲1			▲2		▲3	▲4	

/ACK	D7 to D0	/ACK	D7 to D0	NACK	SP
	▲5		▲6	▲7	▲8

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

▲2: TXI interrupt: Address transmission completed (transfer direction bit: write) (1)

▲3: TXI interrupt: Data transmission completed (first data) (1)

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

▲5: TXI interrupt: Address transmission completed (transfer direction bit: read) (1)

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

▲7: TXI interrupt: Reception for the last data completed (second data) (2)

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

Notes:

1. An interrupt request is generated on the rising edge of the ninth clock.
2. An interrupt request is generated on the rising edge of the eighth clock.

5.3 Operating Test Environment

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

Table 5-6 Operation Test Environment for Rev.1.60 and Rev.1.70.

Item	Contents
Integrated development environment	Renesas Electronics e2 studio V3.1.2.09
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.02.00
	Compiler options: The integrated development environment default settings are used, with the following option added. -lang = c99
Endian order	Big-endian/Little-endian
Module version	Rev.1.60 and Rev.1.70
Board used	Renesas Starter Kit for RX111 (product number. R0K505111SxxxBE) Renesas Starter Kit for RX113 (product number. R0K505113SxxxBE) Renesas Starter Kit for RX231 (product number. R0K505231SxxxBE) Renesas Starter Kit+ for RX63N (product number. R0K50563NSxxxBE) Renesas Starter Kit+ for RX64M (product number. R0K50564MSxxxBE) Renesas Starter Kit+ for RX71M (product number. R0K50571MSxxxBE)

Table 5-7 Operation Test Environment for Rev.1.80.

Item	Contents
Integrated development environment	Renesas Electronics e2 studio V4.0.2.008
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.03.00
	Compiler options: The integrated development environment default settings are used, with the following option added. -lang = c99
Endian order	Big-endian/Little-endian
Module version	Rev.1.80
Board used	Renesas Starter Kit for RX130 (product number. R0K505113SxxxBE) Renesas Starter Kit for RX23T (product number. RTK500523TSxxxxxBE)

Table 5-8 Operation Confirmation Environment for Rev.1.90.

Item	Contents
Integrated development environment	Renesas Electronics e2 studio V4.1.0.018
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.03.00
	Compiler options: The integrated development environment default settings are used, with the following option added. -lang = c99
Endian order	Big-endian/Little-endian
Module version	Rev.1.90
Board used	Renesas Starter Kit for RX111 (product number. R0K505111SxxxBE) Renesas Starter Kit for RX113 (product number. R0K505113SxxxBE) Renesas Starter Kit for RX130 (product number. RTK5005130SxxxxxBE) Renesas Starter Kit for RX231 (product number. R0K505231SxxxBE) Renesas Starter Kit for RX23T (product number. RTK500523TSxxxxxBE) Renesas Starter Kit for RX24T (product number. RTK500524TSxxxxxBE) Renesas Starter Kit for RX63N (product number. R0K50563NSxxxBE) Renesas Starter Kit for RX64M (product number. R0K50564MSxxxBE) Renesas Starter Kit for RX71M (product number. R0K50571MSxxxBE)

Table 5-9 Operation Confirmation Environment for Rev.2.00.

Item	Contents
Integrated deveropment environment	Renesas Electronics e2 studio V5.0.1.005
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.05.00
	Compiler options: The integrated development environment default settings are used, with the following option added. -lang = c99
Endian order	Big-endian/Little-endian
Module version	Rev.2.00
Board used	Renesas Starter Kit for RX231 (product number. R0K505231SxxxBE) Renesas Starter Kit+ for RX65N (product number. RTK500565NSxxxxxBE) Renesas Starter Kit for RX111 (product number. R0K505111SxxxBE) Renesas Starter Kit for RX130 (product number. RTK5005130SxxxxxBE) Renesas Starter Kit for RX231 (product number. R0K505231SxxxBE) Renesas Starter Kit for RX23T (product number. RTK500523TSxxxxxBE) Renesas Starter Kit for RX24T (product number. RTK500524TSxxxxxBE) Renesas Starter Kit+ for RX63N (product number. R0K50563NSxxxBE) Renesas Starter Kit+ for RX64M (product number. R0K50564MSxxxBE) Renesas Starter Kit+ for RX65N (product number. RTK500565NSxxxxxBE) Renesas Starter Kit+ for RX71M (product number. R0K50571MSxxxBE)

Table 5-10 Operation Confirmation Environment for Rev.2.20.

Item	Contents
Integrated deveropment environment	Renesas Electronics e2 studio V6.0.0.001
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.06.00 C/C++ compiler for RX Family V.2.07.00
	Compiler options: The integrated development environment default settings are used, with the following option added. -lang = c99
Endian order	Big-endian/Little-endian
Module version	Rev.2.20
Board used	Renesas Starter Kit for RX24U (product number. RTK50524USxxxxxBE) Renesas Starter Kit for RX130-512KB (product number. RTK5051308SxxxxxBE) Renesas Starter Kit+ for RX65N-2MB (product number. RTK50565N2SxxxxxBE)

5.4 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 e^2 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_sci_iic_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_sci_iic_rx_config.h” may be wrong. Check the file “r_sci_iic_rx_config.h”. If there is a wrong setting, set the correct value for that. Refer to 2.7 Configuration Overview for details.

6. Sample Code

6.1 Example when Accessing One Slave Device Continuously with One Channel

This section describes an example of using one SCI channel in simple I²C mode to continuously write to one slave device.

The procedure is as follows:

1. Execute the R_SCI_IIC_Open function to use SCI channel 1 in the SCI simple I²C mode FIT module.
2. Execute the R_SCI_IIC_MasterSend function to write 3-byte data to device A.
3. Update the transmit data.
4. Execute the R_SCI_IIC_MasterSend function to write 3-byte data to device A.
5. Execute the R_SCI_IIC_Close function to release SCI channel 1 from the SCI simple I²C mode FIT module.

The following abbreviations are used in the program example:
- ST: Start condition
- SP: Stop condition

```
#include <stddef.h> // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

/* Defines the number of retries when a NACK is detected. */
#define RETRY_TMO       10

/* Defines the number of software loops to wait until next communication starts when retrying*/
#define RETRY_WAIT_TIME 1000

/* Transmit size */
#define SEND_SIZE       3

/* Mode definitions in the sample code. */
typedef enum
{
    IDLE = 0U,                  /* Being in idle state */
    BUSY,                       /* I2C communication being performed */
    INITIALIZE,                 /* Simple I2C mode FIT module initialization */
    DEVICE_A_WRITE,             /* Writing device A */
    FINISH,                     /* Communication completed */
    RETRY_WAIT_DEV_A_WR,        /* Waiting for retry writing device A */
    ERROR                       /* Error occurred */
} sample_mode_t;

/* Variable for modes in the sample code */
volatile uint8_t    sample_mode;

/* Variable for the number of retries */
uint32_t            retry_cnt;

/* Variable for the number of transmissions */
uint8_t             send_num = 0;

void main(void);
void Callback_deviceA(void);

void main(void)
{
    sci_iic_return_t   ret;            /* For verifying the return value of the API function */
    volatile uint32_t  retry_wait_cnt = 0;       /* Counter for adjusting the retry interval */

    sci_iic_info_t  iic_info_deviceA;               /* Information structure for device A */
    uint8_t slave_addr_deviceA[1]   = {0x50};       /* Slave address of device A */
    uint8_t access_addr_deviceA[1]  = {0x00};       /* Address to be accessed in device A */
    uint8_t send_data[6]            = {0x81,0x82,0x83,0x84,0x85,0x86};  /* Transmit data */
```

Figure 6.1 Example when Accessing One Slave Device Continuously with One Channel (1/4)

```
        sample_mode = INITIALIZE;                       /* Proceed to initialization processing */

        while(1)
        {
            switch(sample_mode)
            {
                /* Being in idle state */
                case IDLE:
                    /* No operation is performed. */
                break;

                /* I2C communication being performed */
                case BUSY:
                    /* No operation is performed. */
                break;

                /* Initializes the simple I2C mode FIT module. */
                case INITIALIZE:
                    /* Verifies if it is the first time to communicate with device A. */
                    if (0 == send_num)
                    {
                        /* Verifies if channel 1 is currently communicating. */
                        if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[1])
                        {
                            sample_mode = ERROR;        /* Proceed to error processing */
                        }
                        else
                        {
                          /* Configures the device A information structure (transmit pattern 1). */
                            iic_info_deviceA.p_slv_adr = slave_addr_deviceA;
                            iic_info_deviceA.p_data1st = access_addr_deviceA;
                            iic_info_deviceA.p_data2nd = send_data;
                            iic_info_deviceA.dev_sts = SCI_IIC_NO_INIT;
                            iic_info_deviceA.cnt1st = sizeof(access_addr_deviceA);
                            iic_info_deviceA.cnt2nd = SEND_SIZE;
                            iic_info_deviceA.callbackfunc = &Callback_deviceA;
                            iic_info_deviceA.ch_no = 1;
                        }

                        retry_cnt = 0;

                        /* SCI open processing */
                        ret = R_SCI_IIC_Open(&iic_info_deviceA);

                        if (SCI_IIC_SUCCESS == ret)
                        {
                            sample_mode = DEVICE_A_WRITE;   /* Proceed to write processing for
                                                               device A */
                        }
                        else
                        {
                            /* Error processing at the R_SCI_IIC_Open() function call */
                            sample_mode = ERROR;        /* Proceed to error processing */
                        }
                    }
                    /* Verifies if it is the second or the subsequent continuous communication
                       with device A. */
                    else if (1 <= send_num)
                    {
                        /* Verifies if channel 1 is currently communicating. */
                        if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[1])
                        {
                            sample_mode = ERROR;        /* Proceed to error processing */
                        }
                        else
                        {
                           /* Information structure for device A (master transmission pattern 1) */
                            access_addr_deviceA[0] = (access_addr_deviceA[0] + SEND_SIZE);
                            iic_info_deviceA.p_data1st = access_addr_deviceA;
                            iic_info_deviceA.p_data2nd = (send_data + (SEND_SIZE * send_num));
                            iic_info_deviceA.cnt1st = sizeof(access_addr_deviceA);
                            iic_info_deviceA.cnt2nd = SEND_SIZE;
```

A loop is performed with empty processing during idle or I2C communication.

The channel state can be verified with the global variable “g_sci_iic_ChStatus[ ]”.

Initializes the transmit counters and pointers for the second transmission.

Figure 6.2 Example when Accessing One Slave Device Continuously with One Channel (2/4)

```
                    sample_mode = DEVICE_A_WRITE; /* Proceed to write processing for
                                                     device A */
                }
            }
        break;

        /* Writes data to device A */
        case DEVICE_A_WRITE:
            retry_cnt = retry_cnt + 1;

            /* Starts master transmission. */
            ret = R_SCI_IIC_MasterSend(&iic_info_deviceA);

            if (SCI_IIC_SUCCESS == ret)
            {
              sample_mode = BUSY;          /* Then the state becomes "I2C communication
                                              being performed". */
            }
            else if (SCI_IIC_ERR_BUS_BUSY == ret)
            {
                sample_mode = RETRY_WAIT_DEV_A_WR; /* Proceed to a wait for retry */
            }
            else
            {
                /* Error processing at the R_SCI_IIC_MasterSend() function call */
                sample_mode = ERROR;             /* Proceed to error processing */
            }
        break;

        /* Waits for retry writing device A. */
        case RETRY_WAIT_DEV_A_WR:
            retry_wait_cnt = retry_wait_cnt + 1;

            if (RETRY_TMO < retry_cnt)
            {
                retry_wait_cnt = 0;
                sample_mode = ERROR;             /* Proceed to error processing */
            }

            if (RETRY_WAIT_TIME < retry_wait_cnt)
            {
                retry_wait_cnt = 0;

                switch (sample_mode)
                {
                    case RETRY_WAIT_DEV_A_WR:
                        sample_mode = DEVICE_A_WRITE; /* Proceed to write processing for
                                                         device A */
                  break;

                    default:
                        /* No operation is performed. */
                    break;
                }
            }
        break;
```

Processing from ST generation to SP generation is performed by executing the R_SCI_IIC_MasterSend function in the FIT module. After SP is output, the specified callback function (Callback_deviceA()) is called.

When the communication target is the EEPROM, if write operation is performed by sending the write command, a NACK is returned until the write operation is completed.
In the sample code, retry to start communication is performed until an ACK is returned.

Figure 6.3 Example when Accessing One Slave Device Continuously with One Channel (3/4)

```
    /* Communication end processing */
            case FINISH:
                /* SCI close processing */
                ret = R_SCI_IIC_Close(&iic_info_deviceA);

                if (SCI_IIC_SUCCESS == ret)
                {
                    sample_mode = IDLE;             /* Then the state becomes "idle". */
                }
                else
                {
                    /* Error processing at the R_SCI_IIC_Close() function call */
                    sample_mode = ERROR;            /* Proceed to error processing */
                }
            break;

            /* Error occurred */
            case ERROR:
                /* No operation is performed. */
            break;

            default:
                /* No operation is performed. */
            break;
        }
    }
}


void Callback_deviceA(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 1;

    /* Obtains the simple I2C status. */
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* Error processing at the R_SCI_IIC_GetStatus() function call */
        sample_mode = ERROR;                        /* Proceed to error processing */
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
            /* Processing when NACK is detected with the iic_status flag verification. */
            sample_mode = RETRY_WAIT_DEV_A_WR;
        }
        else
        {
            retry_cnt = 0;
            send_num++;
            if (1 >= send_num)
            {
                sample_mode = INITIALIZE;         /* Proceed to initialization processing */
            }
            else
            {
                sample_mode = FINISH;          /* Proceed to communication end processing */
            }
        }
    }
}
```

When the communication has been completed, the SCI channel used can be released by calling the R_SCI_IIC_Close function.
Call the R_SCI_IIC_Close function in the following cases:
- When entering low power consumption mode.
- When communication error occurred.
- When the SCI channel used needs to be released.

Figure 6.4 Example when Accessing One Slave Device Continuously with One Channel (4/4)

6.2 Example when Accessing Two Slave Devices with One Channel

This section describes an example of using one SCI channel in simple I²C mode to write to and read from two slave devices. In the sample code, I²C communication information structure is configured for each accessing device.

The procedure is as follows:

1. Execute the R_SCI_IIC_Open function to use SCI channel 1 in the SCI simple I²C mode FIT module.
2. Execute the R_SCI_IIC_MasterSend function to write 3-byte data to device A.
3. Execute the R_SCI_IIC_MasterReceive function to read 3-byte data from device B.
4. Execute the R_SCI_IIC_Close function to release SCI channel 1 from the SCI simple I²C mode FIT module.

The following abbreviations are used in the program example:
- ST: Start condition
- SP: Stop condition

```
#include <stddef.h> // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

/* Defines the number of retries when a NACK is detected. */
#define RETRY_TMO       10

/* Defines the number of software loops to wait until next communication starts when retrying*/
#define RETRY_WAIT_TIME 1000

/* Transmit size */
#define SEND_SIZE       3

/* Receive size */
#define RECEIVE_SIZE    3

/* Definitions for mode management in the sample code */
typedef enum
{
    IDLE = 0U,                  /* Being in idle state */
    BUSY,                       /* I2C communication being performed */
    INITIALIZE,                 /* Simple I2C mode FIT module initialization */
    DEVICE_A_WRITE,             /* Writing device A */
    DEVICE_B_READ,              /* Reading device B */
    FINISH,                     /* Communication completed */
    RETRY_WAIT_DEV_A_WR,        /* Waiting for retry writing device A */
    RETRY_WAIT_DEV_B_RD,        /* Waiting for retry reading device B */
    ERROR                       /* Error occurred */
} sample_mode_t;

/* Variable for modes in the sample code */
volatile uint8_t                sample_mode;

/* Variable for the number of retries */
volatile uint32_t               retry_cnt;

void main(void);
void Callback_deviceA(void);
void Callback_deviceB(void);

void main(void)
{
  volatile sci_iic_return_t  ret;   /* For verifying the return value of the API function */
  volatile uint32_t          retry_wait_cnt = 0; /* Counter for adjusting the retry interval */

    sci_iic_info_t  iic_info_deviceA;               /* Information structure for device A */
    sci_iic_info_t  iic_info_deviceB;               /* Information structure for device B */
```

Declares information structures as many as devices to communicate.

Figure 6.5 Example when Accessing Two Slave Devices with One Channel (1/5)

```
    uint8_t slave_addr_deviceA[1]   = {0x51};       /* Slave address of device A */
    uint8_t slave_addr_deviceB[1]   = {0x52};       /* Slave address of device B */
    uint8_t access_addr_deviceA[1]  = {0x00};       /* Address to be accessed in device A */
    uint8_t access_addr_deviceB[2]  = {0x00,0x00};  /* Address to be accessed in device B */
    uint8_t send_data[5]            = {0x81,0x82,0x83,0x84,0x85}; /* Transmit data */
    uint8_t store_area[5]           = {0xFF,0xFF,0xFF,0xFF,0xFF}; /* For receive data storage*/

    sample_mode = INITIALIZE;                       /* Proceed to initialization processing */

    while(1)
    {
        switch(sample_mode)
        {
            /* Being in idle state */
            case IDLE:
                /* No operation is performed. */
            break;

            /* I2C communication being performed */
            case BUSY:
                /* No operation is performed. */
            break;
```

A loop is performed with empty processing during idle or I²C communication.

The channel state can be verified with the global variable "g_sci_iic_ChStatus[ ]".

```
            /* Initializes the simple I2C mode FIT module. */
            case INITIALIZE:
                /* Verifies if channel 1 is currently communicating. */
                if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[1])
                {
                    sample_mode = ERROR;            /* Proceed to error processing */
                }
                else
                {
               /* Configures the device A information structure (master transmit pattern 1). */
                    iic_info_deviceA.p_slv_adr = slave_addr_deviceA;
                    iic_info_deviceA.p_data1st = access_addr_deviceA;
                    iic_info_deviceA.p_data2nd = send_data;
                    iic_info_deviceA.dev_sts = SCI_IIC_NO_INIT;
                    iic_info_deviceA.cnt1st = sizeof(access_addr_deviceA);
                    iic_info_deviceA.cnt2nd = SEND_SIZE;
                    iic_info_deviceA.callbackfunc = &Callback_deviceA;
                    iic_info_deviceA.ch_no = 1;

                    /* Configures the device B information structure (master transmit/receive).
*/
                    iic_info_deviceB.p_slv_adr = slave_addr_deviceB;
                    iic_info_deviceB.p_data1st = access_addr_deviceB;
                    iic_info_deviceB.p_data2nd = store_area;
                    iic_info_deviceB.dev_sts = SCI_IIC_NO_INIT;
                    iic_info_deviceB.cnt1st = sizeof(access_addr_deviceB);
                    iic_info_deviceB.cnt2nd = RECEIVE_SIZE;
                    iic_info_deviceB.callbackfunc = &Callback_deviceB;
                    iic_info_deviceB.ch_no = 1;
                }
```

The SCI resource is maintained for each channel. Thus the R_SCI_IIC_Open function is executed only once.

```
                retry_cnt = 0;                      /* Resets the number of retries. */

                /* SCI open processing */
                ret = R_SCI_IIC_Open(&iic_info_deviceA);

                if (SCI_IIC_SUCCESS == ret)
                {
                   sample_mode = DEVICE_A_WRITE; /* Proceed to write processing for device A */
                }
                else
                {
                   /* Error processing at the R_SCI_IIC_Open() function call. */
                   sample_mode = ERROR;             /* Proceed to error processing */
                }

            break;
```

Figure 6.6 Example when Accessing Two Slave Devices with One Channel (2/5)

Processing from ST generation to SP generation is performed by executing the R_SCI_IIC_MasterSend function in the FIT module. After SP is output, the specified callback function (Callback_deviceA()) is called.

```
        /* Writes data to device A. */
        case DEVICE_A_WRITE:
            retry_cnt = retry_cnt + 1;

            /* Starts master transmission. */
            ret = R_SCI_IIC_MasterSend(&iic_info_deviceA);

            if (SCI_IIC_SUCCESS == ret)
            {
                sample_mode = BUSY;           /* Then the state becomes “I2C communication
                                                 being performed”. */
            }
            else if (SCI_IIC_ERR_BUS_BUSY == ret)
            {
                sample_mode = RETRY_WAIT_DEV_A_WR; /* Proceed to a wait for retry */
            }
            else
            {
                /* Error processing at the R_SCI_IIC_MasterSend() function call. */
                sample_mode = ERROR;            /* Proceed to error processing */
            }
        break;
```

Processing from ST generation to SP generation is performed by executing the R_SCI_IIC_MasterReceive function in the FIT module. After SP is output, the specified callback function (Callback_deviceB()) is called.

```
        /* Reads data from device B. */
        case DEVICE_B_READ:
            retry_cnt = retry_cnt + 1;

            /* Starts master transmit/receive. */
            ret = R_SCI_IIC_MasterReceive(&iic_info_deviceB);

            if (SCI_IIC_SUCCESS == ret)
            {
                sample_mode = BUSY;          /* Then the state becomes “I2C communication
                                                being performed”. */
            }
            else if (SCI_IIC_ERR_BUS_BUSY == ret)
            {
                sample_mode = RETRY_WAIT_DEV_B_RD; /* Proceed to a wait for retry */
            }
            else
            {
                /* Error processing at the R_SCI_IIC_MasterReceive() function call. */
                sample_mode = ERROR;            /* Proceed to error processing */
            }
        break;
```

When the communication target is the EEPROM, if write operation is performed by sending the write command, a NACK is returned until the write operation is completed.
In the sample code, retry to start communication is performed until an ACK is returned.

```
        /* Waits for retry writing device A. */
        /* Waits for retry reading device B. */
        case RETRY_WAIT_DEV_A_WR:
        case RETRY_WAIT_DEV_B_RD:
            retry_wait_cnt = retry_wait_cnt + 1;

            if (RETRY_TMO < retry_cnt)
            {
                retry_wait_cnt = 0;
                sample_mode = ERROR;            /* Proceed to error processing */
            }

            if (RETRY_WAIT_TIME < retry_wait_cnt)
            {
                retry_wait_cnt = 0;

                switch (sample_mode)
                {
                   case RETRY_WAIT_DEV_A_WR:
                   sample_mode = DEVICE_A_WRITE; /* Proceed to write processing for device A */
                   break;

                   case RETRY_WAIT_DEV_B_RD:
                    sample_mode = DEVICE_B_READ; /* Proceed to read processing for device B */
                   break;
```

Figure 6.7 Example when Accessing Two Slave Devices with One Channel (3/5)

```
                    default:
                        /* No operation is performed. */
                    break;
                }
            }
        break;

        /* Communication end processing */
        case FINISH:
            /* SCI close processing */
            ret = R_SCI_IIC_Close(&iic_info_deviceA);

            if (SCI_IIC_SUCCESS == ret)
            {
                sample_mode = IDLE;            /* Then the state becomes “idle”. */
            }
            else
            {
                /* Error processing at the R_SCI_IIC_Close() function call */
                sample_mode = ERROR;           /* Proceed to error processing */
            }
        break;

        /* Error occurred */
        case ERROR:
            /* No operation is performed. */
        break;

        default:
            /* No operation is performed. */
        break;
    }
  }
}


void Callback_deviceA(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 1;

    /* Obtains the simple I2C status. */
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* Error processing at the R_SCI_IIC_GetStatus() function call */
        sample_mode = ERROR;                   /* Proceed to error processing */
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
            /* Processing when NACK is detected with the iic_status flag verification */
            sample_mode = RETRY_WAIT_DEV_A_WR;      /* Proceed to a wait for retry */
        }
        else
        {
            retry_cnt = 0;
            sample_mode = DEVICE_B_READ;            /* Proceed to read processing for device B */
        }
    }
}

void Callback_deviceB(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 1;
```

When the communication has been completed, the SCI channel used can be released by calling the R_SCI_IIC_Close function.
Call the R_SCI_IIC_Close function in the following cases:
- When entering low power consumption mode.
- When communication error occurred.
- When the SCI channel used needs to be released.

Figure 6.8 Example when Accessing Two Slave Devices with One Channel (4/5)

```
    /* Obtains the simple I2C status. */
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* Error processing at the R_SCI_IIC_GetStatus() function call */
        sample_mode = ERROR;                        /* Proceed to error processing */
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
            /* Processing when NACK is detected with the iic_status flag verification */
            sample_mode = RETRY_WAIT_DEV_B_RD;      /* Proceed to a wait for retry */
        }
        else
        {
            retry_cnt = 0;
            sample_mode = FINISH;                /* Proceed to communication end processing */
        }
    }
}
```

Figure 6.9 Example when Accessing Two Slave Devices with One Channel (5/5)

6.3 Example when Accessing Two Slave Devices with Two Channels

This section describes an example of using two SCI channels in simple I²C mode to write and read two slave devices. Each channel writes to and reads from different slave device.

In the sample code, I²C communication information structure is configured for each accessing device.

The procedure is as follows:

1. Execute the R_SCI_IIC_Open function to use SCI channel 1 in the SCI simple I²C mode FIT module.
   Also execute the R_SCI_IIC_Open function to use SCI channel 5 in the SCI simple I²C mode FIT module.
2. Execute the R_SCI_IIC_MasterSend function to write 3-byte data to device A using SCI channel 1.

   Execute the R_SCI_IIC_MasterReceive function to read 3-byte data from device B using SCI channel 5.
3. Execute the R_SCI_IIC_Close function to release SCI channel 1 from the SCI simple I²C mode FIT module.
   Also execute the R_SCI_IIC_Close function to release SCI channel 5 from the SCI simple I²C mode FIT module.

The following abbreviations are used in the program example:
- ST: Start condition
- SP: Stop condition

```
#include <stddef.h> /* NULL definition */
#include "platform.h"
#include "r_sci_iic_rx_if.h"

/* Defines the number of retries when a NACK is detected. */
#define RETRY_TMO       10

/* Defines the number of software loops to wait until next communication starts when retrying*/
#define RETRY_WAIT_TIME 1000

/* Transmit size */
#define SEND_SIZE       3

/* Receive size */
#define RECEIVE_SIZE    3

/* Definitions for mode management in the sample code */
typedef enum
{
    IDLE = 0U,                  /* Being in idle state */
    BUSY,                       /* I2C communication being performed */
    INITIALIZE,                 /* Simple I2C mode FIT module initialization */
    DEVICE_A_WRITE,             /* Writing device A */
    DEVICE_B_READ,              /* Reading device B */
    FINISH,                     /* Communication completed */
    RETRY_WAIT_DEV_A_WR,        /* Waiting for retry writing device A */
    RETRY_WAIT_DEV_B_RD,        /* Waiting for retry reading device B */
    ERROR                       /* Error occurred */
} sample_mode_t;

/* Variable for modes in the sample code */
volatile uint8_t                sample_mode_ch1;
volatile uint8_t                sample_mode_ch5;

/* Variable for the number of retries */
volatile uint32_t               retry_cnt_ch1;
volatile uint32_t               retry_cnt_ch5;

void main(void);
void Callback_deviceA(void);
void Callback_deviceB(void);

void main(void)
{
```

Figure 6.10 Example when Accessing Two Slave Devices with Two Channels (1/6)

```
volatile sci_iic_return_t   ret;     /* For verifying the return value of the API function */
volatile uint32_t retry_wait_cnt_ch1 = 0;      /* Counter for adjusting the retry interval */
volatile uint32_t retry_wait_cnt_ch5 = 0;      /* Counter for adjusting the retry interval */

sci_iic_info_t  iic_info_deviceA;                /* Information structure for device A */
sci_iic_info_t  iic_info_deviceB;                /* Information structure for device B */
uint8_t slave_addr_deviceA[1]   = {0x50};        /* Slave address of device A */
uint8_t slave_addr_deviceB[1]   = {0x50};        /* Slave address of device B */
uint8_t access_addr_deviceA[1]  = {0x00};        /* Address to be accessed in device A */
uint8_t access_addr_deviceB[2]  = {0x00,0x00};   /* Address to be accessed in device B */
uint8_t send_data[5]            = {0x81,0x82,0x83,0x84,0x85}; /* Transmit data */
uint8_t store_area[5]           = {0xFF,0xFF,0xFF,0xFF,0xFF}; /* For receive data storage */

sample_mode_ch1 = INITIALIZE;                 /* Ch1: Proceed to initialization processing */
sample_mode_ch5 = INITIALIZE;                 /* Ch5: Proceed to initialization processing */

while(1)
{
    switch(sample_mode_ch1)
    {
        /* Being in idle state */
        case IDLE:
            /* No operation is performed. */
        break;

        /* I2C Communication being performed */
        case BUSY:
            /* No operation is performed. */
        break;

        /* Initializes the simple I2C mode FIT module. */
        case INITIALIZE:
            /* Verifies if channel 1 is currently communicating. */
            if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[1])
            {
                sample_mode_ch1 = ERROR;     /* Ch1: Proceed to error processing */
            }
            else
            {
             /* Configures the device A information structure (master transmit pattern 1). */
                iic_info_deviceA.p_slv_adr = slave_addr_deviceA;
                iic_info_deviceA.p_data1st = access_addr_deviceA;
                iic_info_deviceA.p_data2nd = send_data;
                iic_info_deviceA.dev_sts = SCI_IIC_NO_INIT;
                iic_info_deviceA.cnt1st = sizeof(access_addr_deviceA);
                iic_info_deviceA.cnt2nd = SEND_SIZE;
                iic_info_deviceA.callbackfunc = &Callback_deviceA;
                iic_info_deviceA.ch_no = 1;
            }

            retry_cnt_ch1 = 0;                 /* Resets the number of retries. */

            /* SCI open processing */
            ret = R_SCI_IIC_Open(&iic_info_deviceA);

            if (SCI_IIC_SUCCESS == ret)
            {
                sample_mode_ch1 = DEVICE_A_WRITE; /* Ch1: Proceed to write processing for
                                                     device A */
            }
            else
            {
                /* Error processing at the R_SCI_IIC_Open() function call */
                sample_mode_ch1 = ERROR;        /* Ch1: Proceed to error processing */
            }
        break;

        /* Writes data to device A. */
        case DEVICE_A_WRITE:
            retry_cnt_ch1 = retry_cnt_ch1 + 1;
```

Declares information structures for each device to be accessed.

Processing for different channels can be operated simultaneously. Therefore mode is controlled for each channel.

A loop is performed with empty processing during idle or I²C communication.

The channel state can be verified with the global variable "g_sci_iic_ChStatus[ ]".

Figure 6.11 Example when Accessing Two Slave Devices with Two Channels (2/6)

```
            /* Starts master transmission. */
            ret = R_SCI_IIC_MasterSend(&iic_info_deviceA);

            if (SCI_IIC_SUCCESS == ret)
            {
               sample_mode_ch1 = BUSY;      /* Then the channel 1 state becomes
                                              "I2C communication being performed". */
            }
            else if (SCI_IIC_ERR_BUS_BUSY == ret)
            {
               sample_mode_ch1 = RETRY_WAIT_DEV_A_WR;  /* Ch1: Proceed to a wait for retry */
            }
            else
            {
                /* Error processing at the R_SCI_IIC_MasterSend() function call */
                sample_mode_ch1 = ERROR;        /* Ch1: Proceed to error processing */
            }
        break;

        /* Waits for retry writing device A. */
        case RETRY_WAIT_DEV_A_WR:
            retry_wait_cnt_ch1 = retry_wait_cnt_ch1 + 1;

            if (RETRY_TMO < retry_cnt_ch1)
            {
                retry_wait_cnt_ch1 = 0;
                sample_mode_ch1 = ERROR;        /* Ch1: Proceed to error processing */
            }

            if (RETRY_WAIT_TIME < retry_wait_cnt_ch1)
            {
                retry_wait_cnt_ch1 = 0;

                switch (sample_mode_ch1)
                {
                    case RETRY_WAIT_DEV_A_WR:
                    sample_mode_ch1 = DEVICE_A_WRITE; /* Ch1: Proceed to write processing
                                                         for device A*/
                    break;

                    default:
                        /* No operation is performed. */
                    break;
                }
            }
        break;

        /* Communication end processing */
        case FINISH:
            /* SCI close processing */
            ret = R_SCI_IIC_Close(&iic_info_deviceA);

            if (SCI_IIC_SUCCESS == ret)
            {
                sample_mode_ch1 = IDLE;        /* Then the channel 1 state becomes "idle". */
            }
            else
            {
                /* Error processing at the R_SCI_IIC_Close() function call */
                sample_mode_ch1 = ERROR;        /* Ch1: Proceed to error processing */
            }
        break;

        /* Error occurred */
        case ERROR:
            /* No operation is performed. */
        break;

        default:
            /* No operation is performed. */
        break;
    }
```

Processing from ST generation to SP generation is performed by executing this function. After SP is output, the specified callback function (Callback_deviceA()) is called.

When the communication target is the EEPROM, if write operation is performed by sending the write command, a NACK is returned until the write operation is completed.
In the sample code, retry to start communication is performed until an ACK is returned.

When the communication has been completed, the SCI channel used can be released by calling the R_SCI_IIC_Close function.
Call the R_SCI_IIC_Close function in the following cases:
- When entering low power consumption mode.
- When communication error occurred.
- When the SCI channel used needs to be released.

Figure 6.12 Example when Accessing Two Slave Devices with Two Channels (3/6)

```
        switch(sample_mode_ch5)
         {
            /* Being in idle state */
            case IDLE:
                /* No operation is performed. */
            break;

            /* I2C communication being performed */
            case BUSY:
                /* No operation is performed. */
            break;

            /* Initializes the simple I2C mode FIT module. */
            case INITIALIZE:
                /* Verifies if channel 5 is currently communicating. */
                if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[5])
                {
                    sample_mode_ch5 = ERROR;        /* Ch5: Proceed to error processing */
                }
                else
                {
                    /* Configures the device B information structure (master transmit/receive).
 */
                    iic_info_deviceB.p_slv_adr = slave_addr_deviceB;
                    iic_info_deviceB.p_data1st = access_addr_deviceB;
                    iic_info_deviceB.p_data2nd = store_area;
                    iic_info_deviceB.dev_sts = SCI_IIC_NO_INIT;
                    iic_info_deviceB.cnt1st = sizeof(access_addr_deviceB);
                    iic_info_deviceB.cnt2nd = RECEIVE_SIZE;
                    iic_info_deviceB.callbackfunc = &Callback_deviceB;
                    iic_info_deviceB.ch_no = 5;
                }

                retry_cnt_ch5 = 0;                  /* Resets the number of retries. */

                /* SCI open processing */
                ret = R_SCI_IIC_Open(&iic_info_deviceB);

                if (SCI_IIC_SUCCESS == ret)
                {
                    sample_mode_ch5 = DEVICE_B_READ; /* Ch5: Proceed to read processing for
                                                        device B */
                }
                else
                {
                    /* Error processing at the R_SCI_IIC_Open() function call */
                    sample_mode_ch5 = ERROR;        /* Ch5: Proceed to error processing */
                }

            break;

            case DEVICE_B_READ:
                retry_cnt_ch5 = retry_cnt_ch5 + 1;

                /* Starts master transmit/receive processing. */
                ret = R_SCI_IIC_MasterReceive(&iic_info_deviceB);

                if (SCI_IIC_SUCCESS == ret)
                {
                    sample_mode_ch5 = BUSY;        /* Then the channel 5 state becomes "I2C
                                                      communication being performed". */
                }
                else if (SCI_IIC_ERR_BUS_BUSY == ret)
                {
                    sample_mode_ch5 = RETRY_WAIT_DEV_B_RD; /* Ch5: Proceed to a wait for retry */
                }
                else
                {
                    /* Error processing at the R_SCI_IIC_MasterReceive() function call */
                    sample_mode_ch5 = ERROR;        /* Ch5: Proceed to error processing */
                }
            break;
```

A loop is performed with empty processing during idle or I²C communication.

The channel state can be verified with the global variable "g_sci_iic_ChStatus[ ]".

Processing from ST generation to SP generation is performed by executing this function in the FIT module. After SP is output, the specified callback function (Callback_deviceB()) is called.

Figure 6.13 Example when Accessing Two Slave Devices with Two Channels (4/6)

```
            /* Waits for retry reading device B. */
            case RETRY_WAIT_DEV_B_RD:
                retry_wait_cnt_ch5 = retry_wait_cnt_ch5 + 1;

                 if (RETRY_TMO < retry_cnt_ch5)
                {
                    retry_wait_cnt_ch5 = 0;
                    sample_mode_ch5 = ERROR;        /* Ch5: Proceed to error processing */
                }
                 if (RETRY_WAIT_TIME < retry_wait_cnt_ch5)
                {
                    retry_wait_cnt_ch5 = 0;

                     switch (sample_mode_ch5)
                    {
                        case RETRY_WAIT_DEV_B_RD:
                         sample_mode_ch5 = DEVICE_B_READ; /* Ch5: Proceed to read processing for
                                                             device B */
                        break;
                         default:
                            /* No operation is performed. */
                        break;
                    }
                }
            break;

             /* Communication end processing */
            case FINISH:
                /* SCI close processing */
                ret = R_SCI_IIC_Close(&iic_info_deviceB);

                 if (SCI_IIC_SUCCESS == ret)
                {
                    sample_mode_ch5 = IDLE;         /* Then the channel 5 state becomes "idle". */
                }
                else
                {
                    /* Error processing at the R_SCI_IIC_Close() function call */
                    sample_mode_ch5 = ERROR;        /* Ch5: Proceed to error processing */
                }
            break;

            /* Error occurred. */
            case ERROR:
                /* No operation is performed. */
            break;

            default:
                /* No operation is performed. */
            break;
        }

    }
}


void Callback_deviceA(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 1;

    /* Obtains the simple I2C status. */
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* Error processing at the R_SCI_IIC_GetStatus() function call */
        sample_mode_ch1 = ERROR;                  /* Ch1: Proceed to error processing */
    }
```

When the communication target is the EEPROM, if write operation is performed by sending the write command, a NACK is returned until the write operation is completed.
In the sample code, retry to start communication is performed until an ACK is returned.

When the communication has been completed, the SCI channel used can be released by calling the R_SCI_IIC_Close function.
Call the R_SCI_IIC_Close function in the following cases:
- When entering low power consumption mode.
- When communication error occurred.
- When the SCI channel used needs to be released.

Figure 6.14 Example when Accessing Two Slave Devices with Two Channels (5/6)

```
        else
        {
            if (1 == iic_status.BIT.NACK)
            {
                /* Processing when NACK is detected with the iic_status flag verification. */
                sample_mode_ch1 = RETRY_WAIT_DEV_A_WR;  /* Ch1: Proceed to a wait for retry */
            }
            else
            {
                retry_cnt_ch1 = 0;
                sample_mode_ch1 = FINISH;  /* Ch1: Proceed to communication end processing */
            }
        }
    }

    void Callback_deviceB(void)
    {
        volatile sci_iic_return_t ret;
        sci_iic_mcu_status_t iic_status;
        sci_iic_info_t iic_info_ch;
        iic_info_ch.ch_no = 5;

        /* Obtains the simple I2C status. */
        ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

        if (SCI_IIC_SUCCESS != ret)
        {
            /* Error processing at the R_SCI_IIC_GetStatus() function call. */
            sample_mode_ch5 = ERROR;                     /* Ch5: Proceed to error processing */
        }
        else
        {
            if (1 == iic_status.BIT.NACK)
            {
                /* Processing when NACK is detected with the iic_status flag verification */
                sample_mode_ch5 = RETRY_WAIT_DEV_B_RD;  /* Ch5: Proceed to a wait for retry */
            }
            else
            {
                retry_cnt_ch5 = 0;
                sample_mode_ch5 = FINISH;  /* Ch5: Proceed to communication end processing */
            }

        }
    }
```

Figure 6.15 Example when Accessing Two Slave Devices with Two Channels (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 C/C++ Compiler CC-RX User's Manual (R20UT3248)
The latest version can be downloaded from the Renesas Electronics website.

Related Technical Updates

This module reflects the content of the following technical updates.

None

Website and Support

Renesas Electronics website

http://www.renesas.com

Inquiries

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

REVISION HISTORY	RX Family Application Note Simple I2C Module Using Firmware Integration Technology

Rev.	Date	Description	
		Page	Summary
1.00	July 1, 2013	-	First edition issued
1.10	Nov. 15, 2013	-	Modified return values.
1.20	July 1, 2014	1	Target Device: Added the RX100 Series support.
		4	Table 1.2 Required Memory Size: - Changed all memory sizes in association with additional support for the RX100 Series and additional function for port selection. - Modified the description of the first sentence below the table.
		13	2.3 Supported Toolchains: Updated the toolchain version to v.2.01.
		15	2.6 Configuration Overview (Configuration options in r_sci_iic_rx_config.h (2/2)): Added the configuration option definitions in association with additional function for port selection.
		17	2.9 Adding the FIT Module to Your Project: Modified the description.
1.30	Oct. 1, 2014	-	Added support for the RX64M Group.
		1	Target Device: Changed according to the products supported.
		1	Added the “Related Documents”.
		3	Limitations: Added three limitation items.
		4	Table 1.2 Required Memory Size: Modified the memory sizes for the ROM, Maximum user stack usage, and Maximum interrupt stack usage.
		13	2.2 Software Requirements: Deleted “r_cgc_rx” since this module is independent of the r_cgc_rx.
		14 to 16	Configuration Overview: - Added channel support for the configuration options. Channels supported: 0 to 9, 12 - Added the following configuration option: “SCI_IIC_CFG_PORT_SETTING_PROCESSING”.
		21, 26, 32	3.2 R_SCI_IIC_MasterSend(),3.3 R_SCI_IIC_MasterReceive(), and 3.4 R_SCI_IIC_Close(): Modified the code in the Example sections.
1.40	Dec. 1, 2014	-	Added support for the RX113 Group.
1.50	Dec. 15, 2014	-	Added support for the RX71M Group.
		4	1. Overview: Changed the first item in the Limitations.
		18, 21 to 34	2.7 Parameters: Added the description regarding the limitation of rewriting the structure. ● The description has been also added to the Parameters in sections 3.1 to 3.6 in 3. API Functions.
		22, 27	3.2 R_SCI_IIC_MasterSend() and 3.3 R_SCI_IIC_MasterReceive(): Added ‘(to be updated)’ to the comments for “p_data1st” and “p_data2nd” in the Parameters.
		48	5. Sample Code: Added.
1.60	Feb. 27, 2015	-	Added support for the RX63N Group.
		5	Table 1.2 Required Memory Size: Modified the memory sizes for the ROM, Maximum user stack usage, and Maximum interrupt stack usage.
		14 to 18	Configuration Overview: - Added channel support for the configuration options. Channels supported: 0 to 12

REVISION HISTORY	RX Family Application Note Simple I²C Module Using Firmware Integration Technology

Rev.	Date	Description	
		Page	Summary
1.60	Feb. 27, 2015	Program	Modified the SCI simple I²C mode FIT module due to the software issue [Description] There are errors in the processing to set the clock source (CKS bit in the SMR register) and the bit rate (BRR register) for the on-chip baud rate generator, so the set values may differ from the expected values. [Conditions] When rev.1.50 or an earlier version of the SCI simple I²C mode FIT module is used with RX64M or RX71M, either of the following conditions is met: - Divided-by-3 is selected as the PLL input frequency division ratio (PLIDIV bit in the PLLCR register). - The tenth place of the PLL frequency multiplication factor is 5 (STC bit in the PLLCR register). [Workaround] Use rev. 1.60 or a later version of the SCI simple I²C mode FIT module.
		Program	Modified the SCI simple I²C mode FIT module due to the software issue [Description] When the bit rate is set to low, the program may go into an infinite loop. [Conditions] The following two conditions are met: - Rev.1.50 or an earlier version of the SCI simple I²C mode FIT module is used. - The BRR register value calculated by the sci_iic_set_frequency function is greater than 255. (The bit rate is extremely low compared to PCLKB.) Example: When PCLKB is 60 MHz, the bit rate is set to 200 bps or less. When PCLKB is 300 kHz, the bit rate is set to 1 bps. [Workaround] Use rev. 1.60 or a later version of the SCI simple I²C mode FIT module
1.70	May. 29, 2015	-	Added support for the RX231 Group.
1.80	Oct. 31, 2015	-	Added support for the RX130 Group, RX230 Group, RX23T Group.
		33	Format of 3.5, R_SCI_IIC_GetStatus(), modified

REVISION HISTORY	RX Family Application Note Simple I²C Module Using Firmware Integration Technology

Rev.	Date	Description	
		Page	Summary
1.90	Mar. 4, 2016	-	Added support for the RX24T Group.
		5	Table 1.2 Required Memory Size, changed.
		17, 18	Added description of r_sci_iic_rx_pin_config.h to section 2.6, Configuration Overview.
		-	Changed “master composite” to “master transmit/receive”.
		45	Modified the macro definition of the internal communication information structure api_Mode, which is the I²C protocol operating mode in the communication in progress (master transmit/receive) state, in Table 4.5, States of Flags on State Transitions.
2.00	Oct. 1, 2016	-	Added support for the RX65N Group.
		15	2.6 Configuration Overview: Changed default value of SCI_IIC_CFG_CHi_SSDA_DELAY_SELECT.
		19	Changed code size description from “Table 1.2 Required Memory Size” to “2.7 Code Size.”
2.20	Aug. 31, 2017	-	Added support for the RX24U Group.
		-	Added support for the RX65N-2MB edition.
		-	Added support for the RX130-512KB edition.
		-	Added support for the RX24T-512KB edition.
		1	Related Documents: Added the following document: “Renesas e² studio Smart Configurator User Guide (R20AN0451)”
		16 to 18	2.4. Usage of Interrupt Vector: added.
		20	In "2.7 Configuration Overview ", SCI_IIC_CFG_CHi_INCLUDED describes the important points to be noted for using the compile time setting
			SCI_IIC_CFG_CHi_BITRATE_BPS describes the important points to be noted for bit rate setting.
		21	A notice of bit setting about SCI_IIC_CFG_PORT_SETTING_PROCESSING is added.
		25	2.11. Adding the FIT Module to Your Project: Revised.
		45, 46	4. Pin Settings: added.
		59 to 61	5.3. Operating Test Environment: Added.
		62	5.4. Troubleshooting: Added.
		Program	Changed default value of SCI_IIC_CFG_CH1_INCLUDED.
		Program	Corrected the drive capacity control setting process by r_sci_iic_io_open() function of RX63N, RX64M, RX65N and RX71M.

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

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

1. Handling of Unused Pins

   Handle unused pins in accordance with the directions given under Handling of Unused Pins in the manual.

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

   The state of the product is undefined at the moment when power is supplied.

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

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.
4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.
5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of Microprocessing unit or Microcontroller unit products in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Colophon 6.0