RX Family

I²C Bus Interface (RIIC) Module Using Firmware Integration Technology

R01AN1692EJ0220
Rev. 2.20
Aug. 31, 2017

Introduction

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

Target Device

- RX110, RX111, RX113 Groups
- RX130 Group
- RX230, RX231, RX23T Groups
- RX24T, RX24U Groups
- 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 I²C bus interface module using firmware integration technology (RIIC FIT module (1)) provides a method to transmit and receive data between the master and slave devices using the I²C bus interface (RIIC). The RIIC is in compliance with the NXP I²C-bus (Inter-IC-Bus) interface.

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

Features supported by this module are as follows:

- Master transmission, master reception, slave transmission, and slave reception
- Multi-master configuration that communicates between multiple masters and one slave.
- Communication mode can be standard or fast mode and the maximum communication rate is 400 kbps.
  However channel 0 of RX64M and RX71M supports fast mode plus and the maximum communication rate is 1 Mbps.

Limitations

This module has the following limitations:

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

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

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

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

(5) The module does not support multiple interrupts.

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

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

1.1 RIIC FIT Module

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

1.2 Outline of the API

Table 1.1 lists the API Functions.

Table 1.1 API Functions

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

1.3 Overview of RIIC FIT Module

1.3.1 Specifications of RIIC FIT Module

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

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

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

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

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

3. When multiple RIIC channels are used, the module can control multiple channels. When the device used has multiple channels, simultaneous communication is available using multiple channels.

4. Multiple slave devices with different addresses 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.

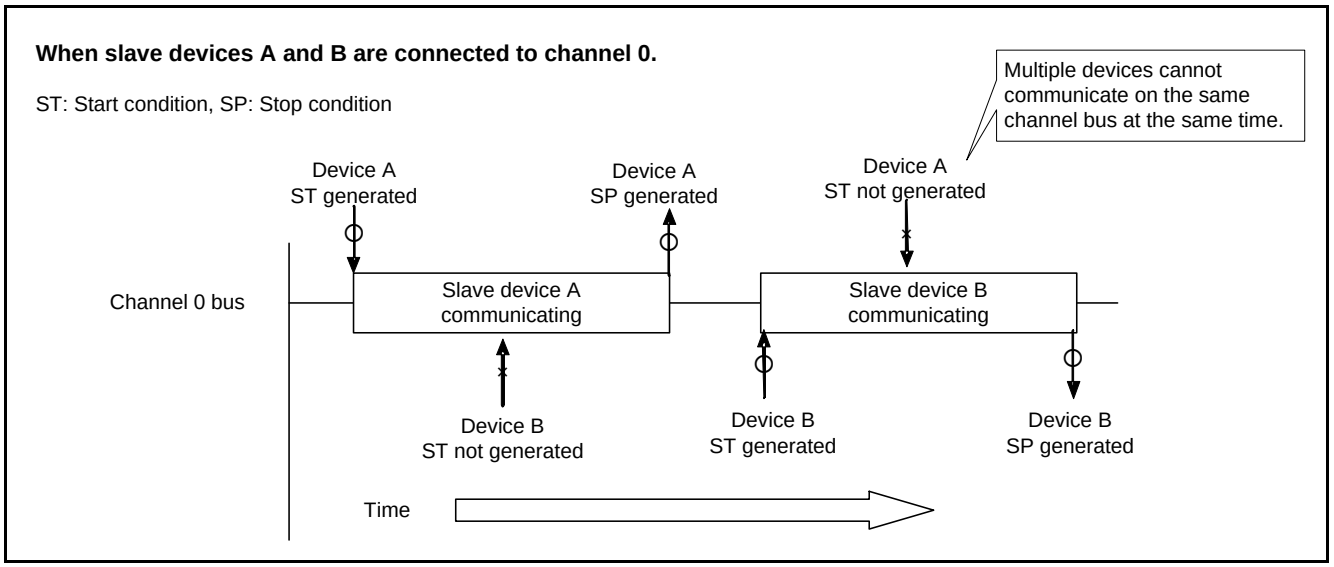


Figure 1.1 Example of Controlling Multiple Slave Devices

1.3.2 Master Transmission

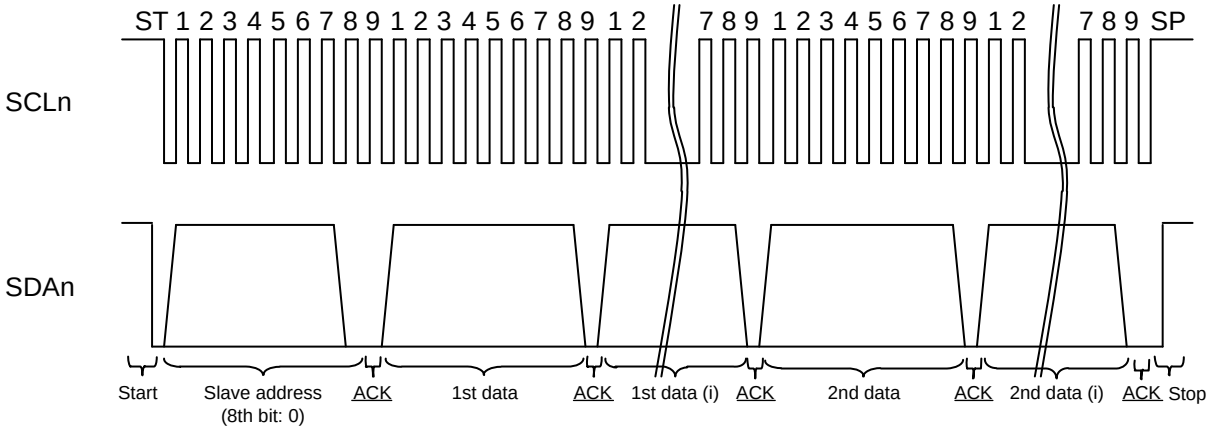
The master device (master (RX MCU)) transmits data to the slave device (slave).

With this module, four patterns of waveforms can be generated for master transmission. A pattern is selected according to the arguments set in the parameters which are members of the I²C communication information structure. Figure 1.2 to Figure 1.5 show the transmit patterns. Refer to 2.9 Parameters for details on the I²C communication information structure.

(1) Pattern 1

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

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



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

Figure 1.2 Signals for Pattern 1 of Master Transmission

(2) Pattern 2

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

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

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

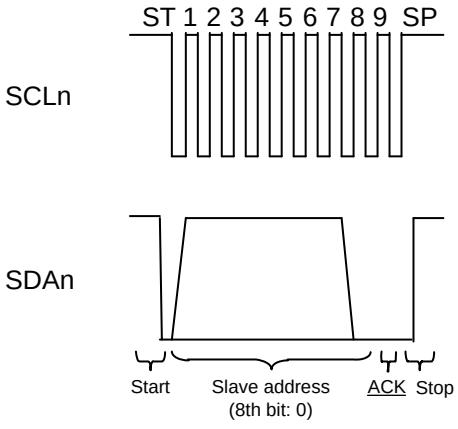
Figure 1.3 Signals for Pattern 2 of Master Transmission

(3) Pattern 3

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

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

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



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

Figure 1.4 Signals for Pattern 3 of Master Transmission

(4) Pattern 4

The master (RX MCU) transmits only a start condition and stop condition to the slave.

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

This pattern is useful for just releasing the bus.

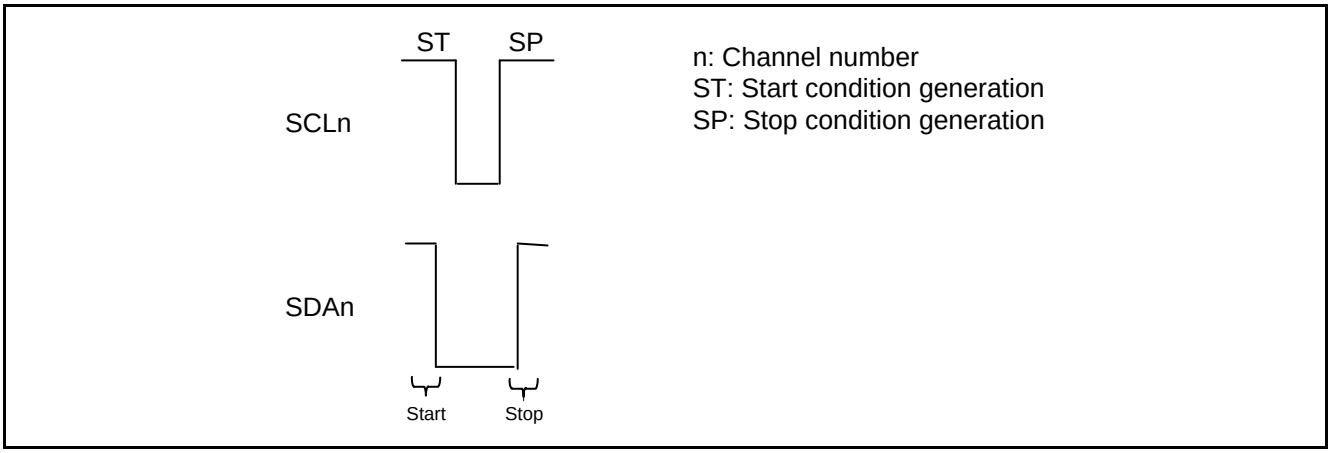


Figure 1.5 Signals for Pattern 4 of Master Transmission

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.

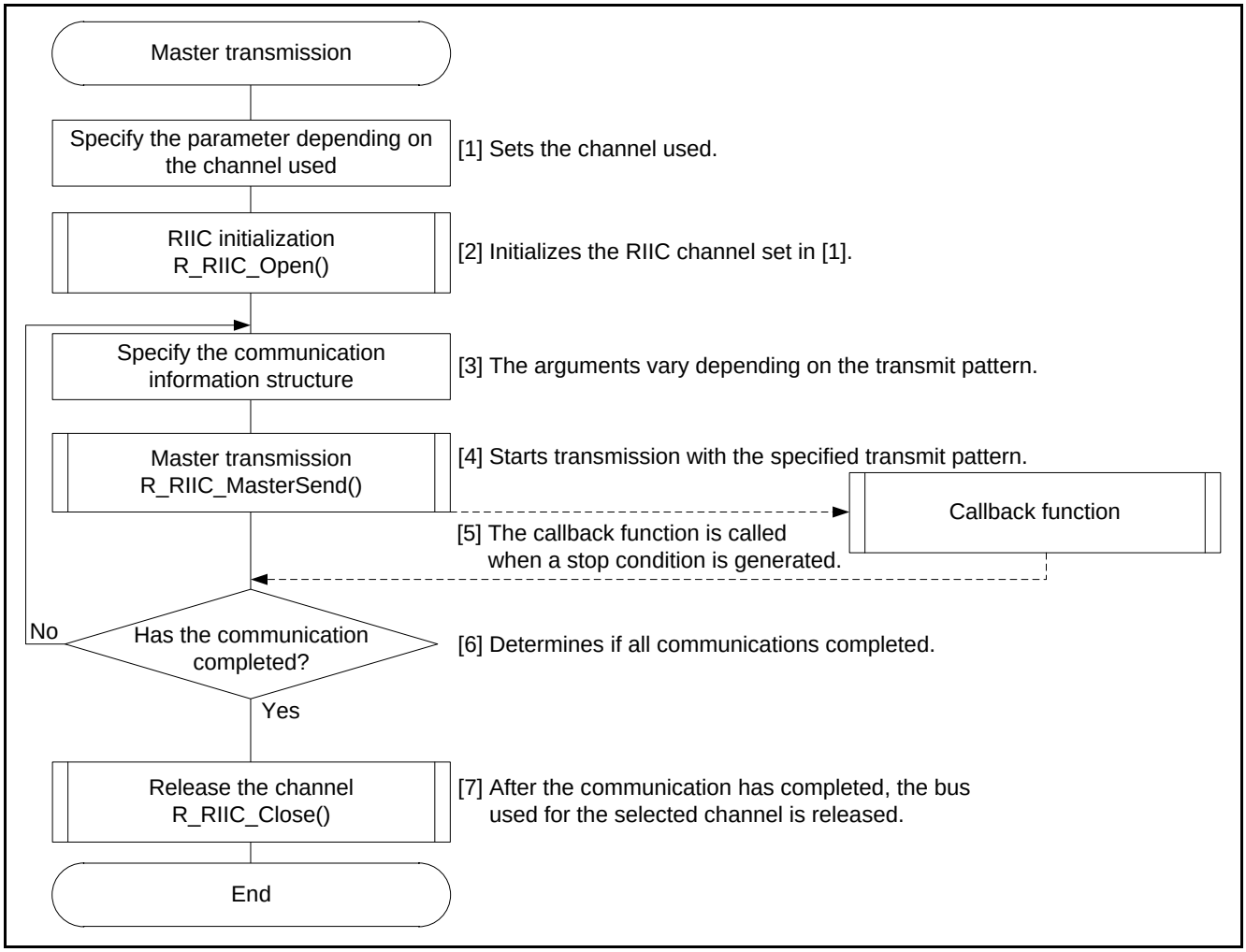


Figure 1.6 Example of Master Transmission

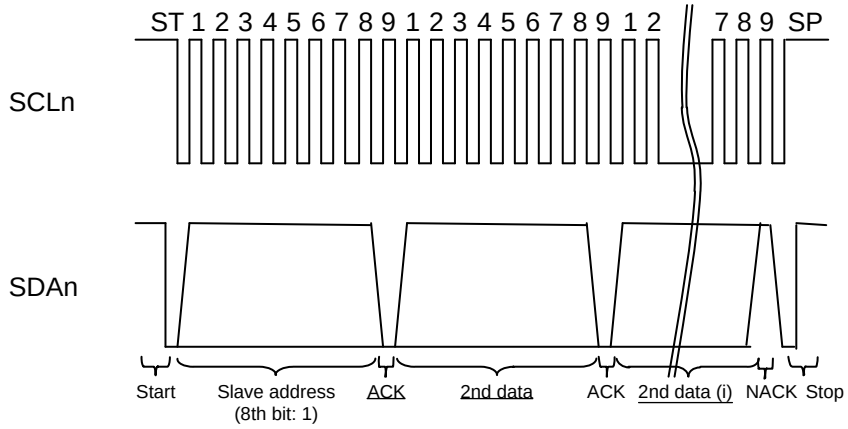
1.3.3 Master Reception

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

(1) Master Reception

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

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



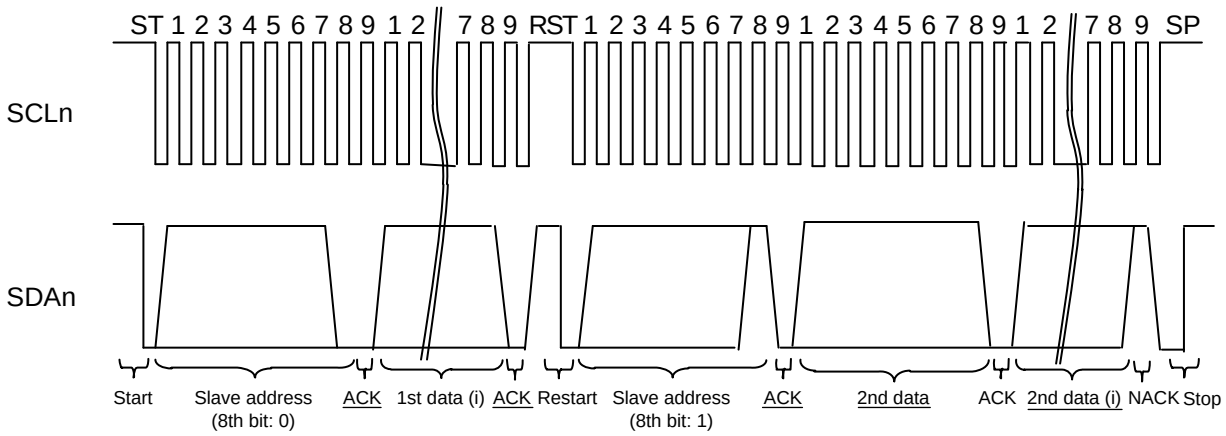
n: Channel number
ST: Start condition generation NACK: Acknowledge: 1
SP: Stop condition generation ACK: Acknowledge: 0
* A signal with an underline indicates data transmission from the slave to the master.

Figure 1.7 Signals for Master Reception

(2) Master Transmit/Receive

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

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



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

Figure 1.8 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 I²C communication information structure member.

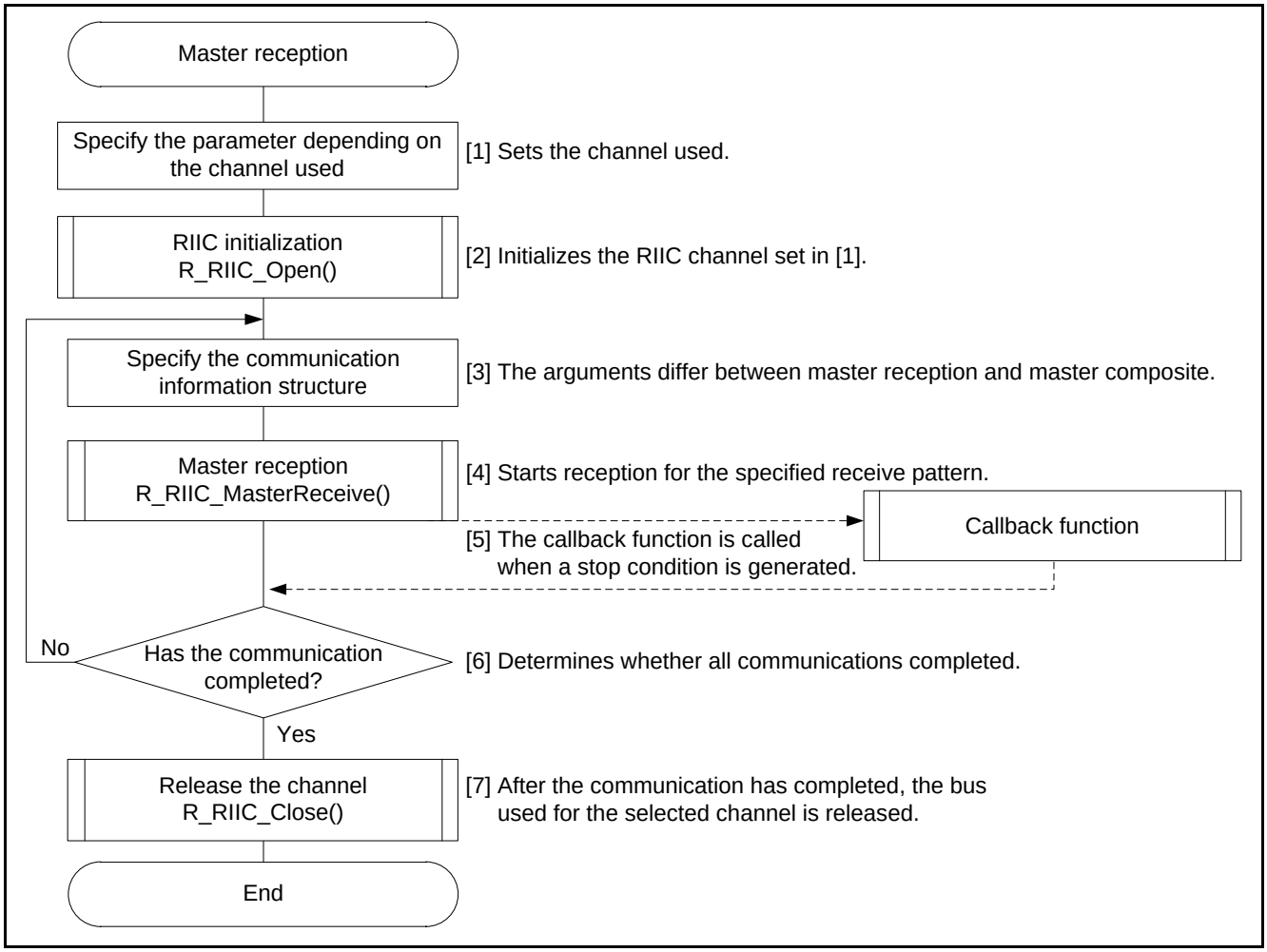


Figure 1.9 Example of Master Reception

1.3.4 Slave Transmission and Reception

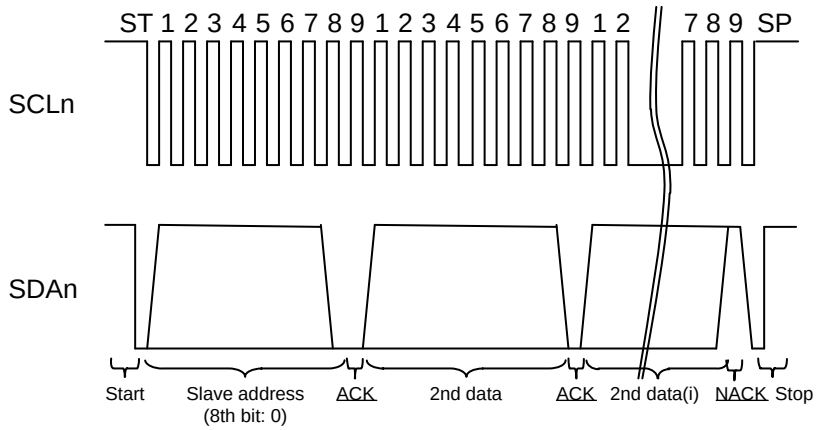
The slave (RX MCU) receives data transmitted from the master. The slave transmits data by the transmit request from the master.

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

(1) Slave Reception

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

After a start condition generated by the master is detected, when the received slave address matches its own address and the eighth bit of the slave address is 0 (write), then the slave starts receive operation. When the last data (the number of data specified in the I²C communication information structure member) is received, a NACK is returned to the master to notify that all necessary data has been received. Figure 1.10 shows the Signals for Slave Reception.



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

Figure 1.10 Signals for Slave Reception

Figure 1.11 shows the procedure of slave reception. The callback function is called after generating a stop condition. Specify the function name in the CallBackFunc of the I²C communication information structure member.

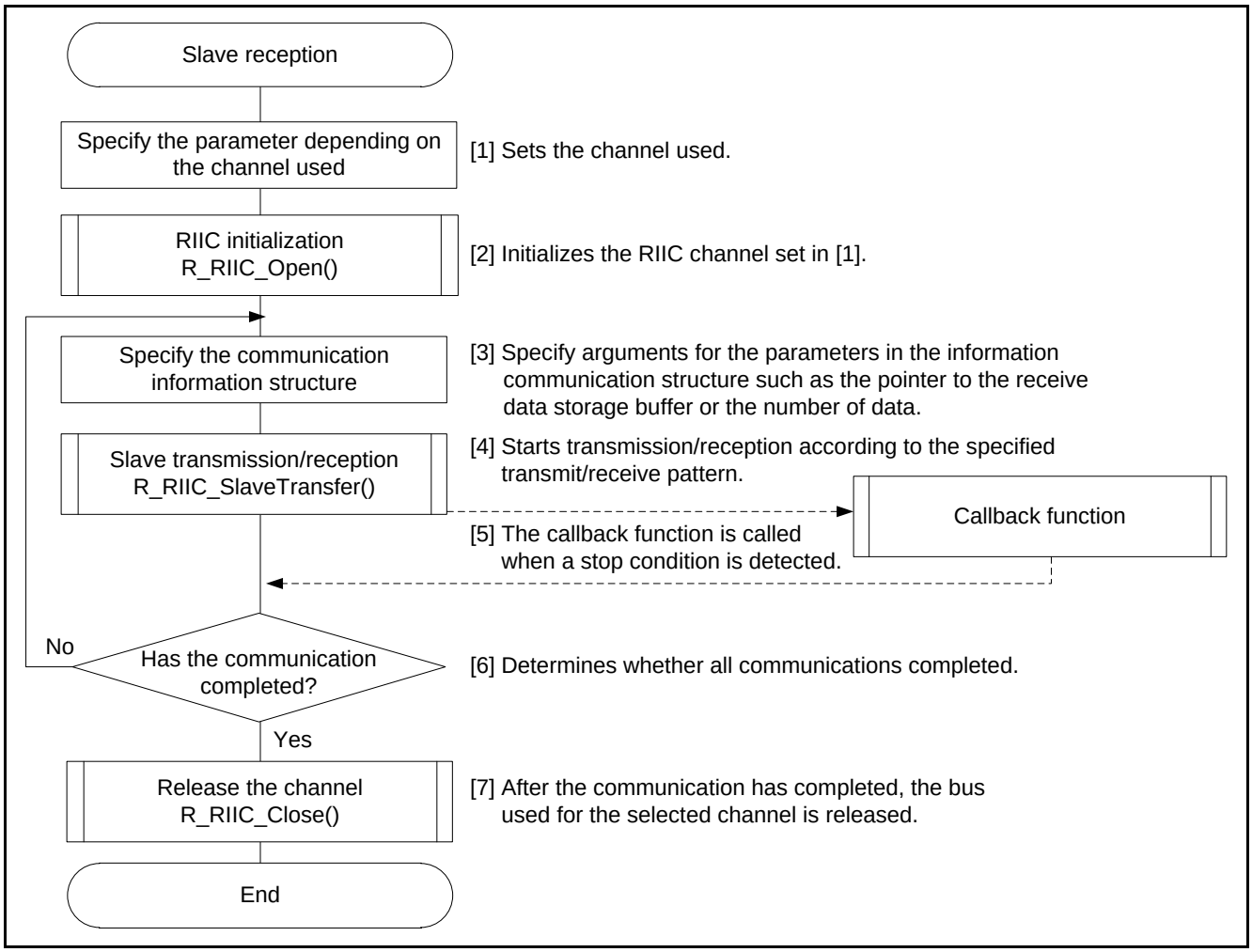
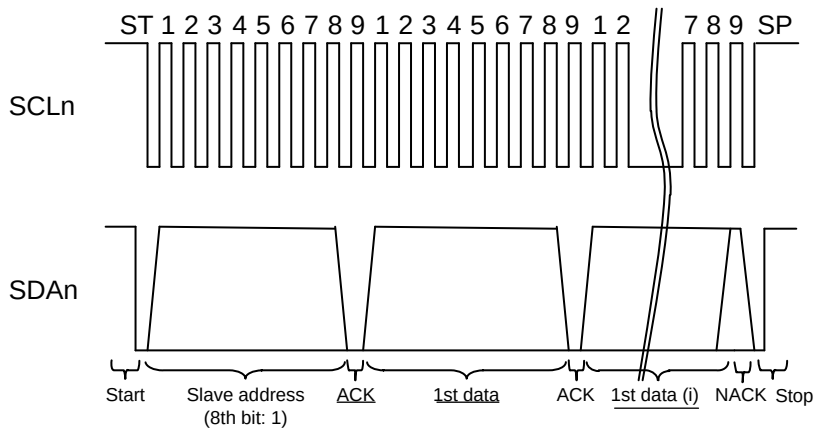


Figure 1.11 Slave Reception

(2) Slave Transmission

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

After a start condition from the master is detected, when the slave address matches its own address and the eighth bit of the slave address is 1 (read), then the slave starts transmit operation. When the transmit request exceeds the number of data specified in the I²C communication information structure member, the slave transmits 0xFF as data. The slave continues transmit operation until a stop condition is detected. Figure 1.12 shows the Signals for Slave Transmission.



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

Figure 1.12 Signals for Slave Transmission

Figure 1.13 shows the procedure of slave transmission. The callback function is called after generating a stop condition. Specify the function name in the CallBackFunc of the I²C communication information structure member.

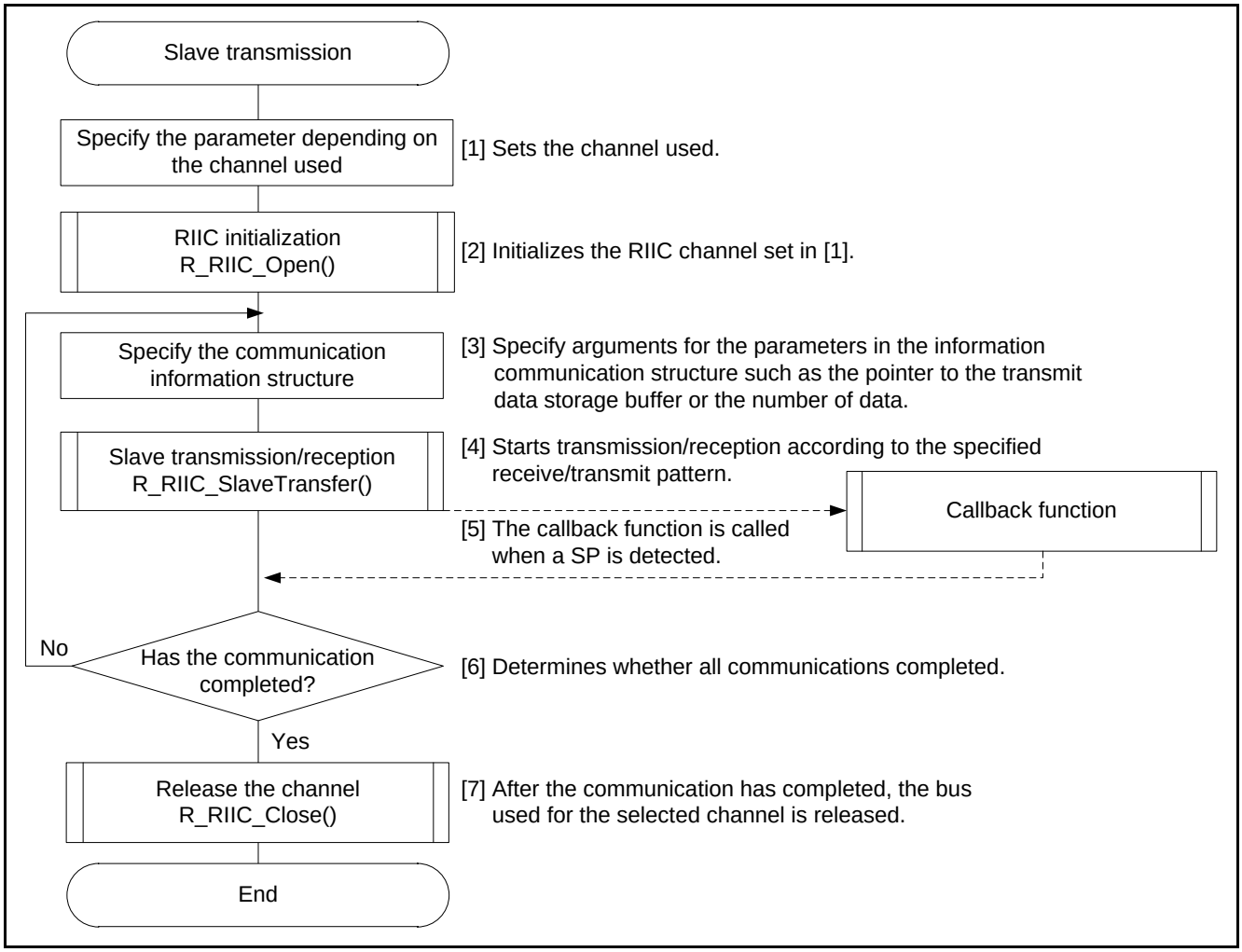


Figure 1.13 Slave Transmission

1.3.5 State Transition

Figure 1.14 shows the RIIC FIT Module State Transition Diagram.

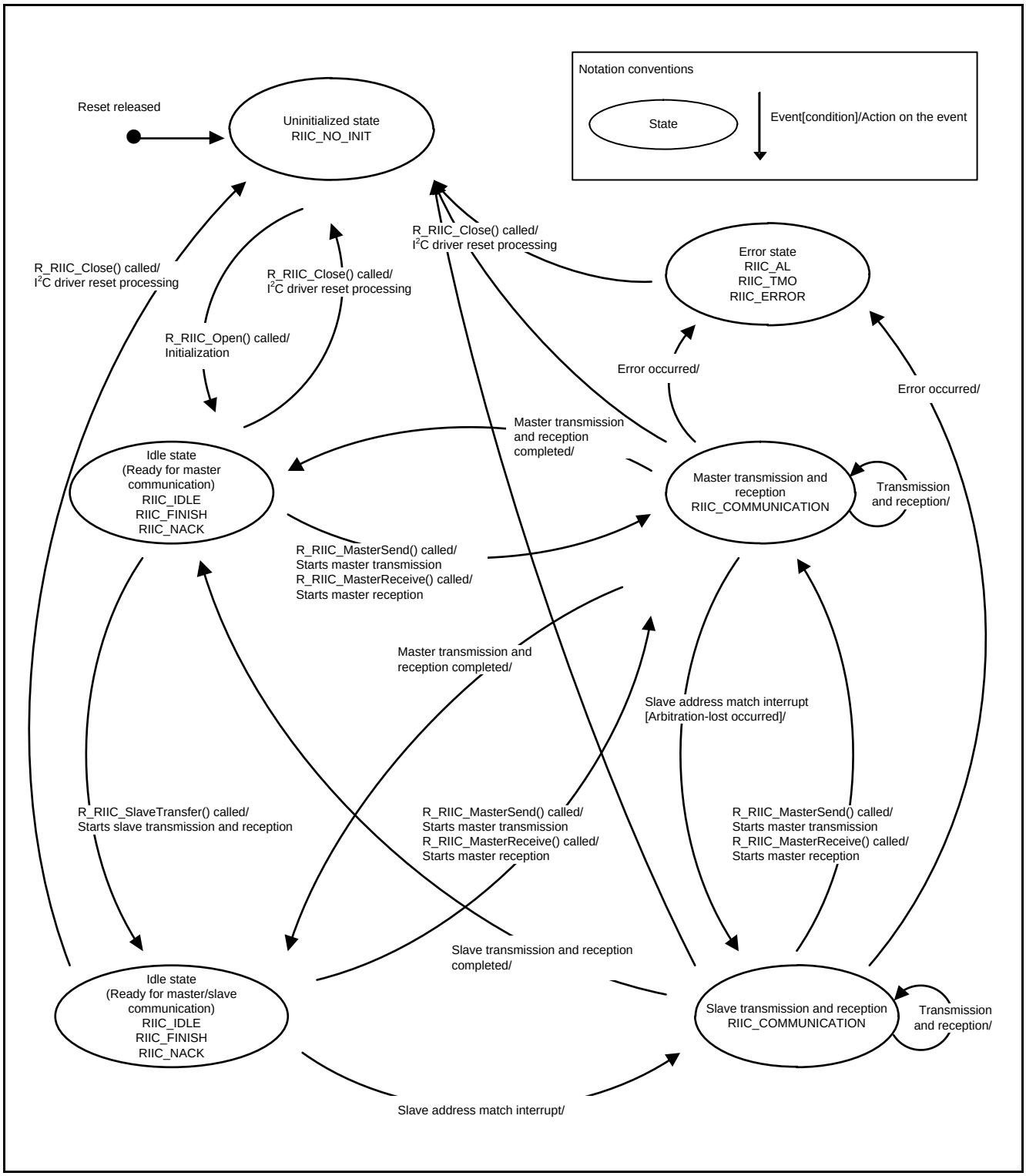


Figure 1.14 RIIC FIT Module State Transition Diagram

1.3.6 Flags when Transitioning States

dev_sts is the device state flag and is one of the I²C communication information structure members. The flag stores the communication state of the device. Using this flag enables controlling multiple slaves on the 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	RIIC_NO_INIT
Idle states	RIIC_IDLE RIIC_FINISH RIIC_NACK
Communicating (master transmission, master reception, slave transmission, and slave reception)	RIIC_COMMUNICATION
Arbitration-lost detection state	RIIC_AL
Timeout detection state	RIIC_TMO
Error	RIIC_ERROR

1.3.7 Arbitration-Lost Detection Function

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

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

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

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

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

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

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

a. When data transmitted by masters are different:

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

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

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

1.3.8 Timeout Detection Function

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

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

(1) The bus is busy in master mode.

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

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

Refer to the following configuration options in "2.7 Configuration Overview" for details on enabling and disabling the timeout detection function.

- RIIC_CFG_CH0_TMO_ENABLE
- RIIC_CFG_CH2_TMO_ENABLE
- RIIC_CFG_CH0_TMO_DET_TIME
- RIIC_CFG_CH2_TMO_DET_TIME
- RIIC_CFG_CH0_TMO_LCNT
- RIIC_CFG_CH2_TMO_LCNT
- RIIC_CFG_CH0_TMO_HCNT
- RIIC_CFG_CH2_TMO_HCNT

Refer to 5.3 Timeout Detection and Processing After the Detection for detailed explanation when a timeout is detected.

2. API Information

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

2.1 Hardware Requirements

This FIT module requires your MCU supports the following feature:

- RIIC

2.2 Software Requirements

This FIT module is dependent upon the following FIT modules:

- Board Support Package Module (r_bsp)

2.3 Supported Toolchains

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

- Renesas RX Toolchain v.2.01.01
- 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 5.4 Operating Test Environment for details.

2.4 Usage of Interrupt Vector

The EEI interrupt, RXI interrupt, TXI interrupt, and TEI interrupt are enabled by execution of R_RIIC_MasterSend function, R_RIIC_MasterReceive function, or R_RIIC_SlaveTransfer function (with specified condition)(while the macro definition RIIC_CFG_CHi_INCLUDE (i = 0 to 2) is 1).

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

Table 2.1 Interrupt Vector used in the RIIC FIT Module

Device	Contents
RX110 RX111 RX113 RX130 RX230 RX231 RX23T RX24T RX24U	EEI0 interrupt [channel 0] (vector no.: 246) RXI0 interrupt [channel 0] (vector no.: 247) TXI0 interrupt [channel 0] (vector no.: 248) TEI0 interrupt [channel 0] (vector no.: 249)
RX64M RX71M	RXI0 interrupt [channel 0] (vector no.: 52) TXI0 interrupt [channel 0] (vector no.: 53) RXI2 interrupt [channel 2] (vector no.: 54) TXI2 interrupt [channel 2] (vector no.: 55) GROUPBL1 interrupt (vector no.: 111) • TEI0 interrupt [channel 0] (group interrupt source no.: 13) • EEI0 interrupt [channel 0] (group interrupt source no.: 14) • TEI2 interrupt [channel 2] (group interrupt source no.: 15) • EEI2 interrupt [channel 2] (group interrupt source no.: 16)
RX65N RX651	RXI0 interrupt [channel 0] (vector no.: 52) TXI0 interrupt [channel 0] (vector no.: 53) RXI1 interrupt [channel 1] (vector no.: 50) TXI1 interrupt [channel 1] (vector no.: 51) RXI2 interrupt [channel 2] (vector no.: 54) TXI2 interrupt [channel 2] (vector no.: 55) GROUPBL1 interrupt (vector no.: 111) • TEI0 interrupt [channel 0] (group interrupt source no.: 13) • EEI0 interrupt [channel 0] (group interrupt source no.: 14) • TEI1 interrupt [channel 1] (group interrupt source no.: 28) • EEI1 interrupt [channel 1] (group interrupt source no.: 29) • TEI2 interrupt [channel 2] (group interrupt source no.: 15) • EEI2 interrupt [channel 2] (group interrupt source no.: 16)

2.5 Header Files

All API calls and their supporting interface definitions are located in r_riic_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_riic_rx_config.h and r_riic_rx_pin_config.h. The option names and setting values are listed in the table below.

Configuration options in *r_riic_rx_config.h* (1/4)	
RIIC_CFG_PARAM_CHECKING_ENABLE **- Default value = 1**	Selects whether to include parameter checking in the code. - When this is set to 0, parameter checking is omitted. With this setting, the code size can be reduced. - When this is set to 1, parameter checking is included.
RIIC_CFG_CHi_INCLUDED (1) **i = 0 to 2** **- When i = 0, the default value = 1** **- When i = 1 to 2, the default value = 0**	Selects whether to use available channels. When not using the channel, set this to 0. - When this is set to 0, relevant processes for the channel are omitted from the code. - When this is set to 1, relevant processes for the channel are included in the code.
RIIC_CFG_CH0_kBPS **- Default value = 400**	Specifies the RIIC0 communication rate. Setting values for the bit rate register and internal reference clock selection bit are calculated using the setting values for RIIC_CFG_CH0_kBPS and the peripheral clock. - Target devices that do not support fast mode plus as the transfer speed. Specify a value less than or equal to 400. - For RX64M ,RX71M and RX65N, specify a value less than or equal to 1000.
RIIC_CFG_CH1_kBPS (1) **- Default value = 400**	Specifies the RIIC1 communication rate. Setting values for the bit rate register and internal reference clock selection bit are calculated using the setting values for RIIC_CFG_CH1_kBPS and the peripheral clock. This should be set to 400 or less.
RIIC_CFG_CH2_kBPS (1) **- Default value = 400**	Specifies the RIIC2 communication rate. Setting values for the bit rate register and internal reference clock selection bit are calculated using the setting values for RIIC_CFG_CH2_kBPS and the peripheral clock. This should be set to 400 or less.
RIIC_CFG_CHi_DIGITAL_FILTER (1) **i = 0 to 2** **- When i = 0 to 2, the default value = 0**	The number of noise filter stage of the specified RIIC channel can be selected. - When this is set to 0, the noise filter is disabled. - When this is set to a value from 1 to 4, values to enable the selected number of filters are selected for the noise filter stage selection bit and digital noise filter circuit enable bit.
RIIC_CFG_PORT_SET_PROCESSING **- Default value = 1**	Specifies whether to include processing for port setting (*) in the code. * Processing for port setting is the setting to use ports selected by R_RIIC_CFG_RIICi_SCLi_PORT, R_RIIC_CFG_RIICi_SCLi_BIT, R_RIIC_CFG_RIICi_SDAi_PORT, and R_RIIC_CFG_RIICi_SDAi_BIT as pins SCL and SDA. - When this is set to 0, processing for port setting is omitted from the code. - When this is set to 1, processing for port setting is included in the code.

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

Configuration options in *r_riic_config.h* (2/4)	
RIIC_CFG_CHi_MASTER_MODE (1) **i = 0 to 2** **- When i = 0 to 2, the default value = 0**	The master arbitration lost detection function of the specified RIIC channel can be enable or disable. Set this to 1 (enabled) when using multi-master. - When this is set to 0, the master arbitration-lost detection is disabled. - When this is set to 1, the master arbitration-lost detection is enabled.
RIIC_CFG_CHi_SLV_ADDR0_FORMAT *1 (1) **RIIC_CFG_CHi_SLV_ADDR1_FORMAT** *2 (1) **RIIC_CFG_CHi_SLV_ADDR2_FORMAT** *2 (1) **i = 0 to 2** ***1: When i = 0 to 2, the default value = 1** ***2: When i = 0 to 2, the default value = 0**	The slave address format can be selected as 7 bits or 10 bits for the specified RIIC channel. - When this is set to 0, the slave address is not set. - When this is set to 1, the 7-bit slave address format is set. - When this is set to 2, the 10-bit slave address format is set.
RIIC_CFG_CHi_SLV_ADDR0 *1 (1) **RIIC_CFG_CHi_SLV_ADDR1** *2 (1) **RIIC_CFG_CHi_SLV_ADDR2** *2 (1) **i = 0 to 2** ***1: When i = 0 to 2, the default value = 0x0025** ***2: When i = 0 to 2, the default value = 0x0000**	This set the slave address of the specified RIIC channel. Available bits of the setting value vary depending on the setting value of the RIIC_CFG_CHi_SLV_ADDRj_FORMAT. (j = 0 to 2) When RIIC_CFG_CH0_SLV_ADDRj_FORMAT is: 0: The setting value is ignored. 1: The lower 7 bits of the setting value are used. 2: The lower 10 bits of the setting value are used.
RIIC_CFG_CHi_SLV_GCA_ENABLE (1) **i = 0 to 2** **- When i = 0 to 2, the default value = 0**	The general call address of the specified RIIC channel can be enable or disable. - When this is set to 0: General call address is disabled. - When this is set to 1: General call address is enabled.
RIIC_CFG_CHi_RXI_INT_PRIORITY (1) **i = 0 to 2** **- When i = 0 to 2, the default value = 1**	The priority level of the receive data full interrupt (RXIi) of the specified RIIC channel can be selected. Specify the level from 1 to 15.
RIIC_CFG_CHi_TXI_INT_PRIORITY (1) **i = 0 to 2** **- When i = 0 to 2, the default value = 1**	The priority level of the transmit data empty interrupt (TXIi) of the specified RIIC channel can be selected. Specify the level from 1 to 15.

Note:

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

Configuration options in *r_riic_config.h* (3/4)	
RIIC_CFG_CHi_EEI_INT_PRIORITY (1) (2) **i = 0 to 2** **- When i = 0 to 2, the default value = 1**	The priority level of the communication error / event occurrence interrupt (EEIi) of the specified RIIC channel can be selected. Specify the level from 1 to 15. Do not set this option to a value lower than the priority level specified with RIIC_CFG_CHi_RXI_INT_PRIORITY or RIIC_CFG_CHi_TXI_INT_PRIORITY.
RIIC_CFG_CHi_TEI_INT_PRIORITY (1) (2) **i = 0 to 2** **- When i = 0 to 2, the default value = 1**	The priority level of the transmission end interrupt (TEIi) of the specified RIIC channel can be selected. Specify the level from 1 to 15. Do not set this option to a value lower than the priority level specified with RIIC_CFG_CHi_RXI_INT_PRIORITY or RIIC_CFG_CHi_TXI_INT_PRIORITY.
RIIC_CFG_CHi_TMO_ENABLE (2) **i = 0 to 2** **- When i = 0 to 2, the default value = 1**	The timeout detection function of the specified RIIC channel can be enabled. - When this is set to 0: RIICi timeout detection function is disabled. - When this is set to 1: RIICi timeout detection function is enabled.
RIIC_CFG_CHi_TMO_DET_TIME (2) **i = 0 to 2** **- When i = 0 to 2, the default value = 0**	You can select the timeout detection time of the specified RIIC channel.- When this is set to 0, long mode is selected. - When this is set to 1, short mode is selected.

Note:

1. The priority level cannot be set individually in devices that group EEI0, TEI0, EEI2, and TEI2 as the BL1 interrupt. In this case, the priority levels for EEI0, TEI0, EEI2, and TEI2 will be unified to all be the maximum value of the individual priority levels set in r_riic_confg.h. However if the other module specifies a greater value than the value specified for the BL1 priority level in the RIIC, the greater value will be used.
   For EEI0 and TEI0 interrupt priority levels, do not set values smaller than the priority levels for RXI0 and TXI0. Also, for EEI2 and TEI2 interrupt priority levels, do not set values smaller than the priority levels for RXI2 and TXI2.
2. This setting is invalid for target devices that do not support the corresponding channel.

Configuration options in *r_riic_config.h* (4/4)	
RIIC_CFG_CHi_TMO_LCNT (1) **i = 0 to 2** **- When i = 0 to 2, the default value = 1**	After enabling the timeout detection function of specified RIIC channel, during the time SCLi line is low, count-up of the internal counter for the timeout detection function can be enabled. - When this is set to 0, counting up is disabled while the SCLi line is held low. - When this is set to 1, counting up is enabled while the SCLi line is held low.
RIIC_CFG_CHi_TMO_HCNT (1) **i = 0 to 2** **- When i = 0 to 2, the default value = 1**	After enabling the specified RIIC timeout detection function, during the time SCLi line is high, the count-up of the internal counter for the timeout detection function can be enabled. - When this is set to 0, counting up is disabled while the SCL0 line is held high. - When this is set to 1, counting up is enabled while the SCL0 line is held high.
RIIC_CFG_BUS_CHECK_COUNTER **- Default value = 1000**	Specifies the timeout counter (number of times to perform bus checking) when the RIIC API function performs bus checking. Specify a value less than or equal to 0xFFFFFFFF. The bus checking is performed in the following timings: - Before generating a start condition - After detecting a stop condition - After generating each condition using the RIIC control function (R_RIIC_Control function) - After generating the SCL one-shot pulse using the RIIC control function (R_RIIC_Control function). With the bus checking, when the bus is busy, the timeout counter is decremented by the software until the bus becomes free. 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. 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) ≈ $(\frac{1}{\text{ICLK}}\ \text{(Hz)}) \times \text{counter value} \times 10$

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

Configuration options in *r_riic_rx_config.h*	
R_RIIC_CFG_RIICi_SCLi_PORT **i = 0 to 2** **- When i = 0, the default value = '1'** **- When i = 1, the default value = '2'** **- When i = 2, the default value = '1'**	Selects port groups used as the SCL pins. Specify the value as an ASCII code in the range '0' to 'J'.
R_RIIC_CFG_RIICi_SCLi_BIT **i = 0 to 2** **- When i = 0, the default value = '2'** **- When i = 1, the default value = '1'** **- When i = 2, the default value = '6'**	Selects pins used as the SCL pins. Specify the value as an ASCII code in the range '0' to '7'.
R_RIIC_CFG_RIICi_SDAi_PORT **i = 0 to 2** **- When i = 0, the default value = '1'** **- When i = 1, the default value = '2'** **- When i = 2, the default value = '1'**	Selects port groups used as the SDA pins. Specify the value as an ASCII code in the range '0' to 'J'.
R_RIIC_CFG_RIICi_SDAi_BIT **i = 0 to 2** **- When i = 0, the default value = '3'** **- When i = 1, the default value = '0'** **- When i = 2, the default value = '7'**	Selects pins used as the SDA pins. 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	9,170 bytes	8,887 bytes	
	RAM	1 channel used	37 bytes		
	Maximum stack usage		396 bytes		Nested interrupts are prohibited, so the maximum value when one channel is used is listed.
RX231	ROM	1 channel used	9,105 bytes	8,822 bytes	
	RAM	1 channel used	37 bytes		
	Maximum stack usage		372 bytes		Nested interrupts are prohibited, so the maximum value when one channel is used is listed.
RX64M	ROM	1 channel used	9,195 bytes	8,912 bytes	
		2 channels used	10,057 bytes	9,774 bytes	
	RAM	1 channel used	111 bytes		
		2 channels used	111 bytes		
	Maximum stack usage		360 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_riic_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 (RIIC_COMMUNICATION).

```
typedef volatile struct
{
   uint8_t rsv2; /* Reserved area */
   uint8_t rsv1; /* Reserved area */
   riic_ch_dev_status_t dev_sts; /* Device state flag */
   uint8_t ch_no; /* Channel number of the used device */
   riic_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 second data storage buffer */
   uint8_t *p_data1st; /* Pointer to the first data storage buffer */
   uint8_t *p_slv_adr; /* Pointer to the slave address storage buffer */
} riic_info_t;
```

2.10 Return Values

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

```
typedef enum
{
   RIIC_SUCCESS = 0U, /* Function processing completed successfully */
   RIIC_ERR_LOCK_FUNC, /* The RIIC is used by another module */
   RIIC_ERR_INVALID_CHAN, /* Nonexistent channel is specified */
   RIIC_ERR_INVALID_ARG, /* Invalid parameter is specified */
   RIIC_ERR_NO_INIT, /* Uninitialized state */
   RIIC_ERR_BUS_BUSY, /* Bus is busy */
   RIIC_ERR_AL, /* The function was called while an arbitration-lost has been detected */
   RIIC_ERR_TMO, /* Timeout is detected */
   RIIC_ERR_OTHER, /* Other error */
} riic_return_t;
```

2.11 Callback Functions

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

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

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

Note:
1. When the timeout detection function is enabled in RIIC_CFG_CHi_TMO_ENABLE (i = 0 to 2) in section 2.7, Configuration Overview.

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

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

2.12 Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends 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_RIIC_Open()

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

Format

```
riic_ return_t R_RIIC_Open(
        riic_info_t *     p_riic_info          /* Structure data */
)
```

Parameters

**p_riic_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 (RIIC_COMMUNICATION) and when an error has occurred (RIIC_TMO and RIIC_ERROR).

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

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

Return Values

RIIC_SUCCESS, / Processing completed successfully */*
RIIC_ERR_LOCK_FUNC, / The API is locked by the other task. */*
RIIC_ERR_INVALID_CHAN, / Nonexistent channel */*
RIIC_ERR_INVALID_ARG, / Invalid parameter */*
RIIC_ERR_OTHER, / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riic_rx_if.h.

Description

Performs the initialization to start the RIIC communication. Sets the RIIC channel specified by the parameter. If the state of the channel is 'uninitialized (RIIC_NO_INIT)', the following processes are performed.

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

Reentrant
Function is reentrant for different channels.

Example

```
volatile riic_return_t  ret;
riic_info_t             iic_info_m;

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

ret = R_RIIC_Open(&iic_info_m);
```

Special Notes
None

3.2 R_RIIC_MasterSend()

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

Format

riic_return_t R_RIIC_MasterSend(

riic_info_t * p_riic_info /* Structure data */

)

Parameters

**p_riic_info*

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

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIIC_COMMUNICATION) and when an error has occurred (RIIC_TMO and RIIC_ERROR).

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

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

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

Return Values

RIIC_SUCCESS / Processing completed successfully */*
RIIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
RIIC_ERR_INVALID_ARG / The parameter is invalid. */*
RIIC_ERR_NO_INIT / Uninitialized state */*
RIIC_ERR_BUS_BUSY / The bus state is busy. */*
RIIC_ERR_AL / Arbitration-lost error occurred */*
RIIC_ERR_TMO / Timeout is detected */*
RIIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riic_rx_if.h.

Description

Starts the RIIC master transmission. The transmission is performed with the RIIC channel and transmit pattern specified by parameters. If the state of the channel is 'idle (RIIC_IDLE, RIIC_FINISH, or RIIC_NACK)', the following processes are performed.

- Setting the state flag
- Initializing variables used by the API
- Enabling the RIIC interrupts

- Generating a start condition

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

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

The transmission processing is performed sequentially in subsequent interrupt processing after this function return RIIC_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_riic_ChStatus [], that is to be "RIIC_FINISH" for normal completion.

Notes:

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

Reentrant

Function is reentrant for different channels.

Example

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

riic_info_t iic_info_m;

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

void main(void)
{
    volatile riic_return_t ret;

    uint8_t addr_eeprom[1]   = {0x50};
    uint8_t access_addr1[1]  = {0x00};
    uint8_t mst_send_data[5] = {0x81,0x82,0x83,0x84,0x85};

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

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

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

    if (RIIC_SUCCESS == ret)
    {
        while(RIIC_FINISH != iic_info_m.dev_sts);
    }
    else
    {
        /* error */
    }

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

void CallbackMaster(void)
{
    volatile riic_return_t ret;
    riic_mcu_status_t      iic_status;

    ret = R_RIIC_GetStatus(&iic_info_m, &iic_status);
    if(RIIC_SUCCESS != ret)
    {
        /* Call error processing for the R_RIIC_GetStatus() function */
    }
    else
    {
        /* Processing when a timeout, arbitration-lost, NACK,
           or others 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	Pointer to the slave address storage buffer			FIT_NO_PTR (1)
*p_data1st	Pointer to the first data storage buffer for transmitting	FIT_NO_PTR (1)	FIT_NO_PTR (1)	FIT_NO_PTR (1)
*p_data2nd	Pointer to the second data storage buffer for transmitting		FIT_NO_PTR (1)	FIT_NO_PTR (1)
cnt1st	0000 0001h to FFFF FFFFh (2)	0	0	0
cnt2nd	0000 0001h to FFFF FFFFh (2)		0	0
callbackfunc	Specify the function name used			
ch_no	00h to FFh			
dev_sts	Device state flag			
rsv1, rsv2	Reserved (value set here has no effect)			

Notes:

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

3.3 R_RIIC_MasterReceive()

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

Format

riic_return_t R_RIIC_MasterRecive(

riic_info_t * p_riic_info /* Structure data */

)

Parameters

**p_riic_info*

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

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIIC_COMMUNICATION) and when an error has occurred (RIIC_TMO and RIIC_ERROR).

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

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

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

Return Values

RIIC_SUCCESS / Processing completed successfully */*
RIIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
RIIC_ERR_INVALID_ARG / The parameter is invalid. */*
RIIC_ERR_NO_INIT / Uninitialized state */*
RIIC_ERR_BUS_BUSY / The bus state is busy. */*
RIIC_ERR_AL / Arbitration-lost error occurred */*
RIIC_ERR_TMO / Timeout is detected */*
RIIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riic_rx_if.h.

Description

Starts the RIIC master reception. The reception is performed with the RIIC channel and receive pattern specified by parameters. If the state of the channel is ‘idle (RIIC_IDLE, RIIC_FINISH, or RIIC_NACK)’, the following processes are performed.

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

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

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

The reception processing is performed sequentially in subsequent interrupt processing after this function return RIIC_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_riic_ChStatus [], that is to be "RIIC_FINISH" for normal completion.

Notes:

1. When SCL and SDA pin is not external pull-up, this function may return RIIC_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>
#include "platform.h"
#include "r_riic_rx_if.h"

riic_info_t    iic_info_m;

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

void main(void)
{
    volatile riic_return_t ret;

    uint8_t addr_eeprom[1]    = {0x50};
    uint8_t access_addr1[1]   = {0x00};
    uint8_t mst_store_area[5] = {0xFF,0xFF,0xFF,0xFF,0xFF};

    /* Sets IIC Information. */
    iic_info_m.dev_sts = RIIC_NO_INIT;
    iic_info_m.ch_no = 0;
```

```
    iic_info_m.callbackfunc = &CallbackMaster;
    iic_info_m.cnt2nd = 3;
    iic_info_m.cnt1st = 1;
    iic_info_m.p_data2nd = mst_store_area;
    iic_info_m.p_data1st = access_addr1;
    iic_info_m.p_slv_adr = addr_eeprom;

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

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

    if (RIIC_SUCCESS == ret)
    {
        while(RIIC_FINISH != iic_info_m.dev_sts);
    }
    else
    {
        /* error */
    }

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

void CallbackMaster(void)
{
    volatile riic_return_t ret;
    riic_mcu_status_t      iic_status;

    ret = R_RIIC_GetStatus(&iic_info_m, &iic_status);
    if(RIIC_SUCCESS != ret)
    {
        /* Call error processing for the R_RIIC_GetStatus() function */
    }
    else
    {
        /* Processing when a timeout, arbitration-lost, NACK,
           or others 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	Pointer to the slave address storage buffer	
*p_data1st	Not used (value set here has no effect)	Pointer to the first data storage buffer for transmitting
*p_data2nd	Pointer to the second data storage buffer for receiving	
dev_sts	Device state flag	
cnt1st (1)	0	0000 0001h to FFFF FFFFh
cnt2nd	0000 0001h to FFFF FFFFh (2)	
callbackfunc	Specify the function name used	
ch_no	00h to FFh	
rsv1, rsv2, rsv3	Reserved (value set here has no effect)	

Notes:

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

3.4 R_RIIC_SlaveTransfer()

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

Format

riic_return_t R_RIIC_SlaveTransfer(

riic_info_t * p_riic_info /* Structure data */

)

Parameters

**p_riic_info*

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

The contents of the structure are referred and updated during communication. Do not rewrite the structure during communication (RIIC_COMMUNICATION) and when an error has occurred (RIIC_TMO and RIIC_ERROR).

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

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

Return Values

RIIC_SUCCESS / Processing completed successfully */*
RIIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
RIIC_ERR_INVALID_ARG / The parameter is invalid. */*
RIIC_ERR_NO_INIT / Uninitialized state */*
RIIC_ERR_BUS_BUSY / The bus state is busy. */*
RIIC_ERR_AL / Arbitration-lost error occurred */*
RIIC_ERR_TMO / Timeout is detected */*
RIIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riic_rx_if.h.

Description

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

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

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

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

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

The interrupt generation timing of slave transmission should be refered from "5.2.4 Slave Transmission". The interrupt generation timing for slave reception should be refered from "5.2.5 Slave reception".

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

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

Reentrant

Function is reentrant for different channels.

Example

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

riic_info_t    iic_info_m;

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

void main(void)
{
    volatile    riic_return_t ret;
    riic_info_t iic_info_s;

    uint8_t addr_eeprom[1]    = {0x50};
    uint8_t access_addr1[1]   = {0x00};
    uint8_t mst_send_data[5]  = {0x81,0x82,0x83,0x84,0x85};
    uint8_t slv_send_data[5]  = {0x71,0x72,0x73,0x74,0x75};
    uint8_t mst_store_area[5] = {0xFF,0xFF,0xFF,0xFF,0xFF};
    uint8_t slv_store_area[5] = {0xFF,0xFF,0xFF,0xFF,0xFF};
```

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

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

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

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

    /* RIIC master send start */
    ret = R_RIIC_MasterSend(&iic_info_m);

    while(1);
}

void CallbackMaster(void)
{
    volatile riic_return_t ret;
    riic_mcu_status_t      iic_status;

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

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

Special Notes

The table below lists available settings for each receive pattern.

Structure Member	Available Parameter Settings	
	Slave Reception	Slave Transmission
*p_slv_adr	Not used (value set here has no effect)	
*p_data1st	(For slave transmission)	Pointer to the first data storage buffer for transmitting (1)
*p_data2nd	Pointer to the second data storage buffer for receiving (2)	(For slave reception)
dev_sts	Device state flag	
cnt1st	(For slave transmission)	0000 0001h to FFFF FFFFh
cnt2nd	0000 0001h to FFFF FFFFh	(For slave reception)
callbackfunc	Specify the function name used	
ch_no	00h to FFh	
rsv1, rsv2, rsv3	Reserved (value set here has no effect)	

Notes:

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

3.5 R_RIIC_GetStatus()

Returns the state of this module.

Format

```
riic_sts_flg_t R_RIIC_GetStatus(
        riic_info_t *     p_riic_info         /* Structure data */
        riic_mcu_status_t *  p_riic_status /* RIIC state */
)
```

Parameters

**p_riic_info*

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

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

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

**p_riic_status*

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

```
typedef union
{
   uint32_t LONG;
   struct
   {
         uint32_t rsv:19;  /* reserve */
         uint32_t TMO:1; /* Timeout flag */
         uint32_t AL:1; /* Arbitration lost detection flag */
         uint32_t rsv:4; /* reserve */
         uint32_t SCLO:1; /* SCL pin output control status */
         uint32_t SDAO:1; /* SDA pin output control status */
         uint32_t SCLI:1; /* SCL pin level */
         uint32_t SDAI:1; /* SDA pin level */
         uint32_t NACK:1; /* NACK detection flag */
         uint32_t rsv:1; /* reserve */
         uint32_t BSY:1; /* Bus status flag */
   }BIT;
} riic_mcu_status_t;
```

Return Values

RIIC_SUCCESS / Processing completed successfully */*
RIIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
RIIC_ERR_INVALID_ARG / The parameter is invalid. */*

Properties
Prototyped in r_riic_rx_if.h.

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

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

Reentrant
Function is reentrant for different channels.

Example

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

iic_info_m.ch_no = 0;

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

Special Notes
The following shows the state flag allocation.

b31 to b16
Reserved
Reserved
Rsv
Undefined

b15 to b13	b12	b11	b10 to b8
Reserved	Event detection		Reserved
Reserved	Timeout detection	Arbitration lost detection	Reserved
Rsv	TMO	AL	Rsv
Undefined	0: Not detected 1: Detected		Undefined

b7	b6	b5	b4	b3	b2	b1	b0
Reserved	Pin status		Pin level		Event detection	Reserved	Bus state
Reserved	SCL pin control	SDA pin control	SCL pin level	SDA pin level	NACK detection	Reserved	Bus busy/ready
rsv	SCLO	SDAO	SCLI	SDAI	NACK	rsv	BSY
Undefined	0: Output low level 1: Output Hi-Z		0: Low level 1: High level		0: Not detected 1: Detected	Undefined	0: Idle 1: Busy

3.6 R_RIIC_Control()

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

Format

```
riic_return_t R_RIIC_Control(
        r_riic_info_t *  p_riic_info          /* Structure data */
        uint8_t  ctrl_ptn             /* Output pattern */
);
```

Parameters

**p_riic_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 (RIIC_COMMUNICATION) and when an error has occurred (RIIC_TMO and RIIC_ERROR).

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

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

ctrl_ptn

Specifies the output pattern.

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

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

- RIIC_GEN_START_CON
- RIIC_GEN_RESTART_CON
- RIIC_GEN_STOP_CON

The following two can specified simultaneously.

- RIIC_GEN_SDA_HI_Z
- RIIC_GEN_SCL_ONESHOT

```
#define RIIC_GEN_START_CON (uint8_t)(0x01) /* Start condition generation */
#define RIIC_GEN_STOP_CON (uint8_t)(0x02) /* Stop condition generation */
#define RIIC_GEN_RESTART_CON (uint8_t)(0x04) /* Restart condition generation */
#define RIIC_GEN_SDA_HI_Z (uint8_t)(0x08) /* Hi-Z output from the SDA pin */
#define RIIC_GEN_SCL_ONESHOT (uint8_t)(0x10) /* SCL clock one-shot output */
#define RIIC_GEN_RESET (uint8_t)(0x20) /* RIIC module reset */
```

Return Values

RIIC_SUCCESS / Processing completed successfully */*
RIIC_ERR_INVALID_CHAN / Nonexistent channel */*
RIIC_ERR_INVALID_ARG / Invalid parameter */*
RIIC_ERR_BUS_BUSY / Bus is busy */*
RIIC_ERR_AL / Arbitration-lost error occurred */*
RIIC_ERR_OTHER / The event occurred is invalid in the current state. */*

Properties

Prototyped in r_riic_rx_if.h.

Description

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

Reentrant

Function is reentrant for different channels.

Example

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

iic_info_m.ch_no = 0;

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

Special Notes

One-shot output of the SCL clock

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

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

3.7 R_RIIC_Close()

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

Format

riic_return_t R_RIIC_Close(

riic_info_t * p_riic_info /* Structure data */

)

Parameters

**p_riic_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 (RIIC_COMMUNICATION) and when an error has occurred (RIIC_TMO and RIIC_ERROR).

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

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

Return Values

RIIC_SUCCESS / Processing completed successfully */*
RIIC_ERR_INVALID_CHAN / The channel is nonexistent. */*
RIIC_ERR_INVALID_ARG / Invalid parameter */*

Properties

Prototyped in r_riic_rx_if.h.

Description

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

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

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

Reentrant

Function is reentrant for different channels.

Example

```
volatile riic_return_t  ret;
riic_info_t             iic_info_m;

iic_info_m.ch_no = 0;

ret = R_RIIC_Close(&iic_info_m);
```

Special Notes

None

3.8 R_RIIC_GetVersion()

Returns the current version of this module.

Format

uint32_t R_RIIC_GetVersion(void)

Parameters

None

Return Values

Version number

Properties

Prototyped in r_riic_rx_if.h.

Description

This function will return the version of the currently installed RIIC 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_RIIC_GetVersion();
```

Special Notes

This function is inlined using '#pragma inline'.

4. Pin Settings

To use the RIIC 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 RIIC FIT module can choose whether or not to perform the pin setting in the R_RIIC_Open function depending on the setting of the configuration option RIIC_CFG_PORT_SET_PROCESSING.

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

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

Table 4.1 Macro Definitions for the Pin Setting Feature

Channel Selected	Pin Selected	Macro Definition
Channel 0	SCL0 Pin	R_RIIC_CFG_RIIC0_SCL0_PORT R_RIIC_CFG_RIIC0_SCL0_BIT
	SDA0 Pin	R_RIIC_CFG_RIIC0_SDA0_PORT R_RIIC_CFG_RIIC0_SDA0_BIT
Channel 1	SCL1 Pin	R_RIIC_CFG_RIIC1_SCL1_PORT R_RIIC_CFG_RIIC1_SCL1_BIT
	SDA1 Pin	R_RIIC_CFG_RIIC1_SDA1_PORT R_RIIC_CFG_RIIC1_SDA1_BIT
Channel 2	SCL2 Pin	R_RIIC_CFG_RIIC2_SCL2_PORT R_RIIC_CFG_RIIC2_SCL2_BIT
	SDA2 Pin	R_RIIC_CFG_RIIC2_SDA2_PORT R_RIIC_CFG_RIIC2_SDA2_BIT

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

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

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

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

5. Appendices

5.1 Communication Method

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

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 r_riic_api_status_t)

No.	Constant Name	Description
STS0	RIIC_STS_NO_INIT	Uninitialized state
STS1	RIIC_STS_IDLE	Idle state (ready for master communication)
STS2	RIIC_STS_IDLE_EN_SLV	Idle state (ready for master/slave communication)
STS3	RIIC_STS_ST_COND_WAIT	Wait state for a start condition to be detected
STS4	RIIC_STS_SEND_SLVADR_W_WAIT	Wait state for the slave address [write] transmission to complete
STS5	RIIC_STS_SEND_SLVADR_R_WAIT	Wait state for the slave address [read] transmission to complete
STS6	RIIC_STS_SEND_DATA_WAIT	Wait state for the data transmission to complete
STS7	RIIC_STS_RECEIVE_DATA_WAIT	Wait state for the data reception to complete
STS8	RIIC_STS_SP_COND_WAIT	Wait state for a stop condition to be detected
STS9	RIIC_STS_AL	Arbitration-lost state
STS10	RIIC_STS_TMO	Timeout detection state

5.1.2 Events During API Operation

Table 5.2 lists the Events Used for Protocol Control. In this module, not only interrupt but also the module function call is defined as event.

Table 5.2 Events Used for Protocol Control (enum r_riic_api_event_t)

No.	Event	Event Definition
EV0	RIIC_EV_INIT	R_RIIC_Open() called
EV1	RIIC_EV_EN_SLV_TRANSFER	R_RIIC_SlaveTransfer() called
EV2	RIIC_EV_GEN_START_COND	R_RIIC_MasterSend() or R_RIIC_MasterReceive() called
EV3	RIIC_EV_INT_START	EEI interrupt occurred (interrupt flag: START)
EV4	RIIC_EV_INT_ADD	TEI interrupt occurred, TXI interrupt occurred
EV5	RIIC_EV_INT_SEND	TEI interrupt occurred, TXI interrupt occurred
EV6	RIIC_EV_INT_RECEIVE	RXI interrupt occurred
EV7	RIIC_EV_INT_STOP	EEI interrupt occurred (interrupt flag: STOP)
EV8	RIIC_EV_INT_AL	EEI interrupt occurred (interrupt flag: AL)
EV9	RIIC_EV_INT_NACK	EEI interrupt occurred (interrupt flag: NACK)
EV10	RIIC_EV_INT_TMO	EEI interrupt occurred (interrupt flag: TMO)

5.1.3 Protocol State Transitions

In this module, a state transition occurs when an interface function provided is called or when an I²C interrupt request is generated. Figure 5.1 to Figure 5.4 show protocol state transitions.

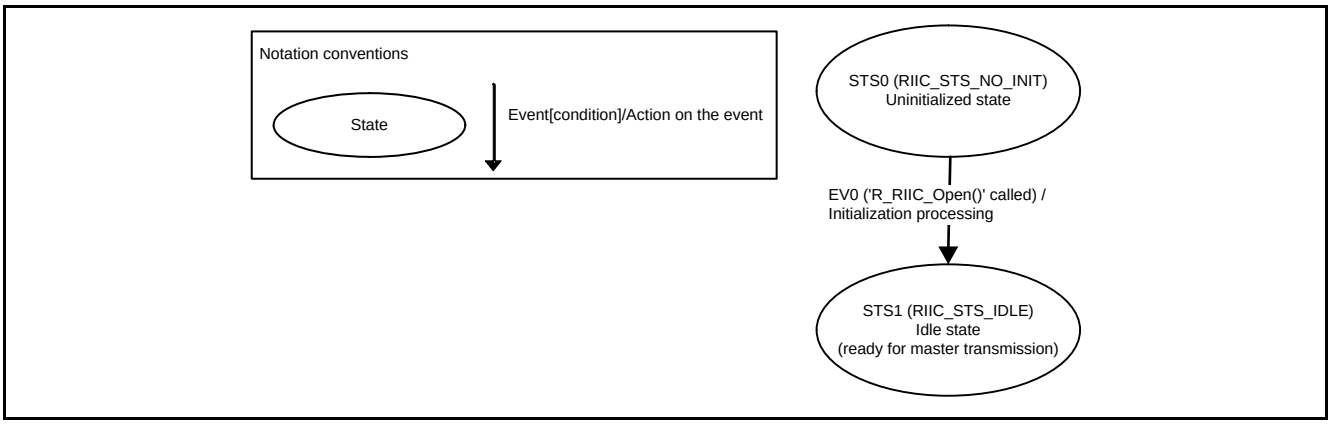


Figure 5.1 State Transition on Initialization ('R_RIIC_Open()' Called)

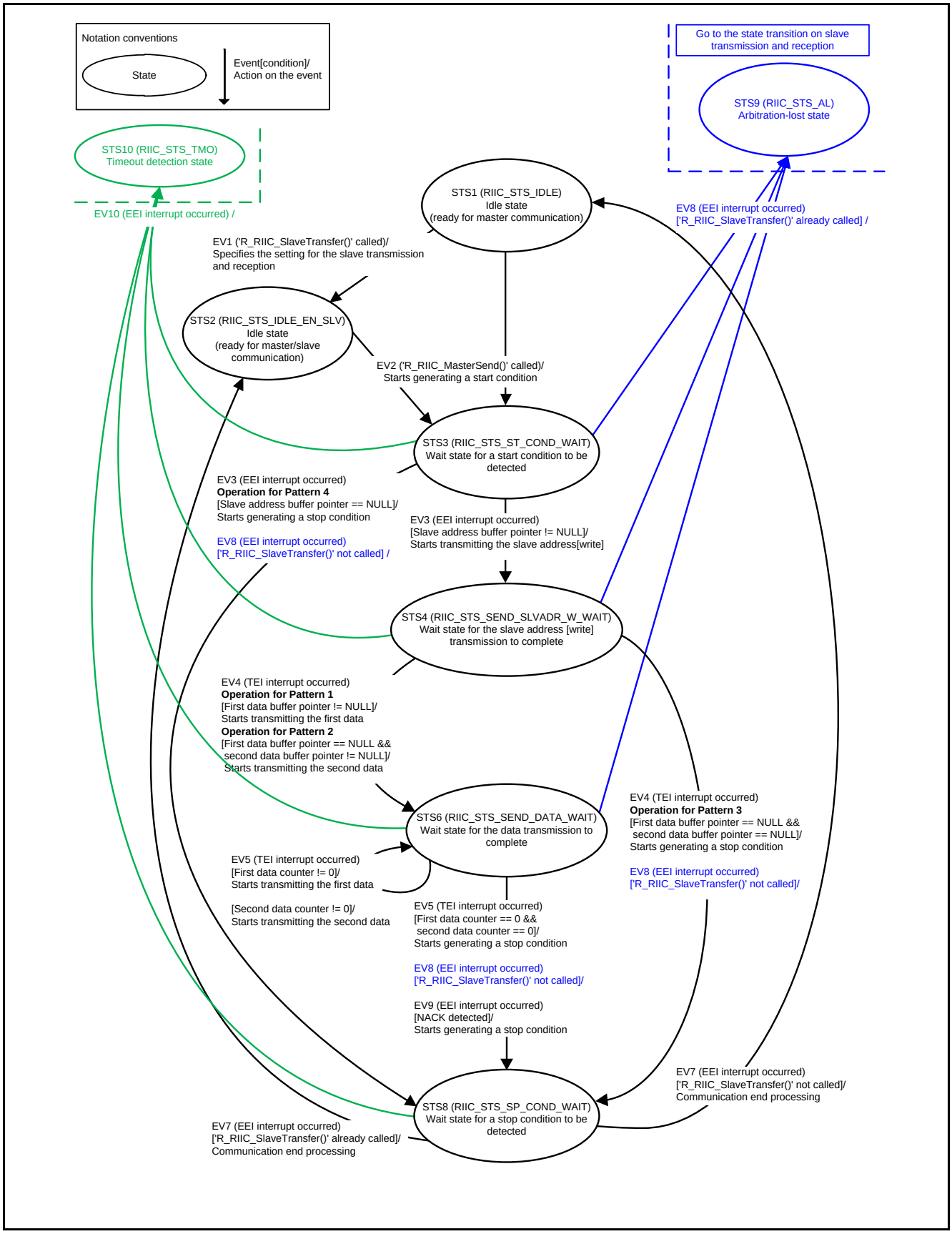


Figure 5.2 State Transition on Master Transmission (R_RIIC_MasterSend() Called)

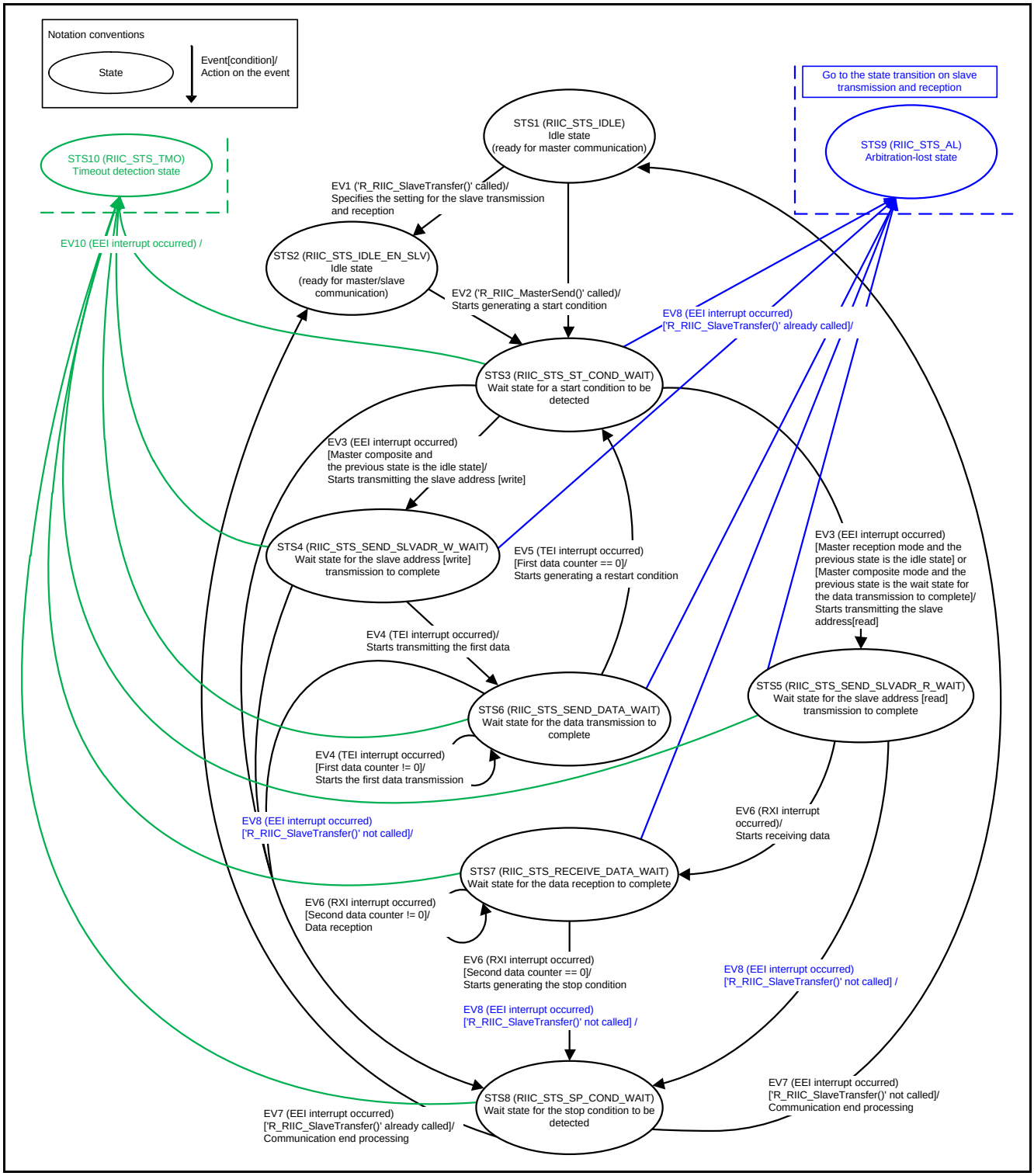


Figure 5.3 State Transition on Master Reception (R_RIIC_MasterReceive() Called)

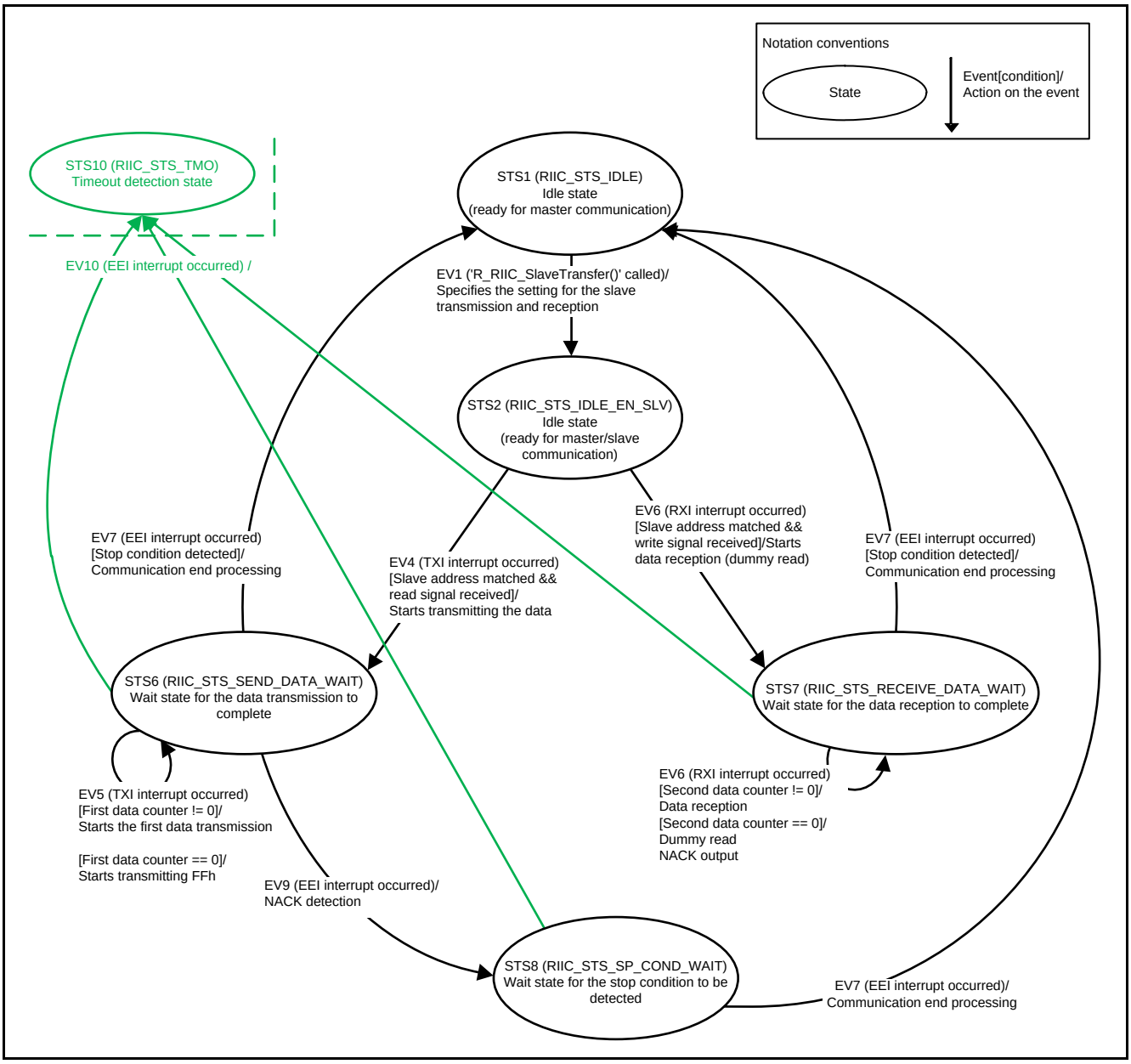


Figure 5.4 State Transition on Slave Transmission and Reception (R_RIIC_SlaveTransfer() Called)

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_riic_mtx_tbl[][]) (1)

State		Event										
		EV0	EV1	EV2	EV3	EV4	EV5	EV6	EV7	EV8	EV9	EV10
STS0	Uninitialized state [RIIC_STS_NO_INIT]	Func0	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
STS1	Idle state (ready for master communication) [RIIC_STS_IDLE]	ERR	Func 10	Func1	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
STS2	Idle state (ready for master/slave communication) [RIIC_STS_IDLE_EN_SLV]	ERR	ERR	Func1	ERR	Func4	ERR	Func4	ERR	ERR	ERR	ERR
STS3	Wait state for the start condition to be generated [RIIC_STS_ST_COND_WAIT]	ERR	ERR	ERR	Func2	ERR	ERR	ERR	ERR	Func8	Func9	Func 11
STS4	Wait state for the slave address [write] to complete [RIIC_STS_SEND_SLVADR_W_WAIT]	ERR	ERR	ERR	ERR	Func3	ERR	ERR	ERR	Func8	Func9	Func 11
STS5	Wait state for the slave address [read] to complete [RIIC_STS_SEND_SLVADR_R_WAIT]	ERR	ERR	ERR	ERR	ERR	ERR	Func3	ERR	Func8	Func9	Func 11
STS6	Wait state for the data transmission to complete [RIIC_STS_SEND_DATA_WAIT]	ERR	ERR	ERR	ERR	ERR	Func5	ERR	ERR	Func8	Func9	Func 11
STS7	Wait state for the data reception to complete [RIIC_STS_RECEIVE_DATA_WAIT]	ERR	ERR	ERR	ERR	ERR	ERR	Func6	ERR	ERR	Func9	Func 11
STS8	Wait state for the stop condition to be generated [RIIC_STS_SP_COND_WAIT]	ERR	ERR	ERR	ERR	ERR	ERR	ERR	Func7	ERR	Func9	Func 11
STS9	Arbitration-lost state [RIIC_STS_AL]	ERR	ERR	ERR	ERR	ERR	Func5	Func6	Func7	ERR	ERR	ERR
STS10	Timeout detection state [RIIC_STS_TMO]	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR

Note:

1. ERR indicates RIIC_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	riic_init_driver()	Initialization
Func1	riic_generate_start_cond()	Start condition generation (for master transmission)
Func2	riic_after_gen_start_cond()	Processing after generating a start condition
Func3	riic_after_send_slvadr()	Processing after completing the slave address transmission
Func4	riic_after_receive_slvadr()	Processing after matching the received slave address
Func5	riic_write_data_sending()	Data transmission
Func6	riic_read_data_receiving()	Data reception
Func7	riic_after_dtct_stop_cond ()	Communication end processing
Func8	riic_arbitration_lost()	Processing when detecting an arbitration-lost
Func9	riic_nack()	Processing when detecting a NACK
Func10	riic_enable_slave_transfer()	Enabling slave transmission/reception
Func11	riic_time_out()	Processing when detecting a timeout

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_riic_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 ‘RIIC_IDLE/RIIC_FINISH/RIIC_NACK’ (idle state (ready for communication)) and communication is available. When the bus is being used for communication, the flag becomes ‘RIIC_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	Device State Flag (Communication Device)	I²C Protocol Operating Mode	Current State of the Protocol Control
	g_riic_ChStatus[]	I²C Communication Information Structure dev_sts	Internal Communication Information Structure N_Mode	Internal Communication Information Structure N_status
Uninitialized state	RIIC_NO_INIT	RIIC_NO_INIT	RIIC_MODE_NONE	RIIC_STS_NO_INIT
Idle state (ready for master communication)	RIIC_IDLE RIIC_FINISH RIIC_NACK	RIIC_IDLE RIIC_FINISH RIIC_NACK	RIIC_MODE_NONE	RIIC_STS_IDLE
Idle state (ready for master/slave communication)	RIIC_IDLE	RIIC_IDLE	RIIC_MODE_S_READY	RIIC_STS_IDLE_EN_SLV
Communicating (master transmission)	RIIC_COMMUNICATION	RIIC_COMMUNICATION	RIIC_MODE_M_SEND	RIIC_STS_ST_COND_WAIT RIIC_STS_SEND_SLVADR_W_WAIT RIIC_STS_SEND_DATA_WAIT RIIC_STS_SP_COND_WAIT RIIC_STS_AL RIIC_STS_TMO
Communicating (master reception)	RIIC_COMMUNICATION	RIIC_COMMUNICATION	RIIC_MODE_ M_RECEIVE	RIIC_STS_ST_COND_WAIT RIIC_STS_SEND_SLVADR_R_WAIT RIIC_STS_RECEIVE_DATA_WAIT RIIC_STS_SP_COND_WAIT RIIC_STS_AL RIIC_STS_TMO
Communicating (master transmit/receive)	RIIC_COMMUNICATION	RIIC_COMMUNICATION	RIIC_MODE_ M_SEND_RECEIVE	RIIC_STS_ST_COND_WAIT RIIC_STS_SEND_SLVADR_W_WAIT RIIC_STS_SEND_SLVADR_R_WAIT RIIC_STS_SEND_DATA_WAIT RIIC_STS_RECEIVE_DATA_WAIT RIIC_STS_SP_COND_WAIT RIIC_STS_AL RIIC_STS_TMO
Communicating (slave transmission)	RIIC_COMMUNICATION	RIIC_COMMUNICATION	RIIC_MODE_S_SEND	RIIC_STS_SEND_DATA_WAIT RIIC_STS_SP_COND_WAIT RIIC_STS_TMO
Communicating (slave reception)	RIIC_COMMUNICATION	RIIC_COMMUNICATION	RIIC_MODE_S_RECEIVE	RIIC_STS_RECEIVE_DATA_WAIT RIIC_STS_SP_COND_WAIT RIIC_STS_TMO
Arbitration-lost detection state	RIIC _AL	RIIC _AL	—	—
Timeout detection state	RIIC_TMO	RIIC_TMO	—	—
Error state	RIIC_ERROR	RIIC_ERROR	—	—

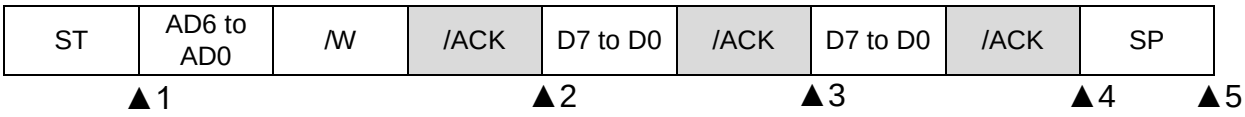
5.2 Interrupt Request Generation Timing

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

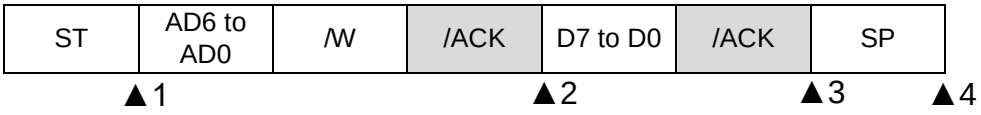
5.2.1 Master Transmission

(1) Pattern 1



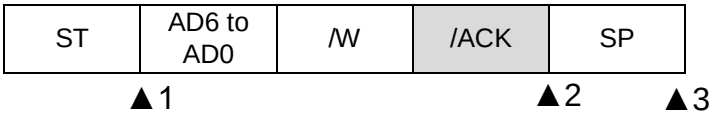
▲1: EEI (START) interrupt: Start condition detected
▲2: TEI interrupt: Address transmission completed (transfer direction bit: write)
▲3: TEI interrupt: Data transmission completed (first data)
▲4: TEI interrupt: Data transmission completed (second data)
▲5: EEI (STOP) interrupt: Stop condition detected

(2) Pattern 2



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

(3) Pattern 3

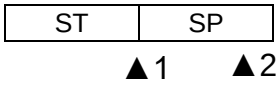


▲1: EEI (START) interrupt: Start condition detected

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

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

(4) Pattern 4



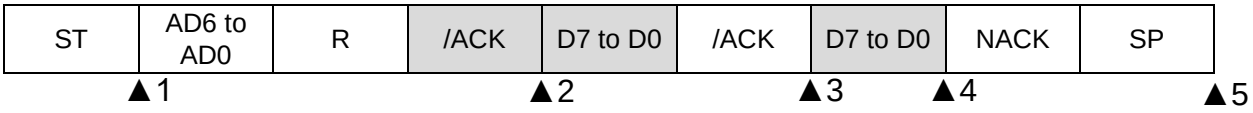
▲1: EEI (START) interrupt: Start condition detected

▲2: EEI (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



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

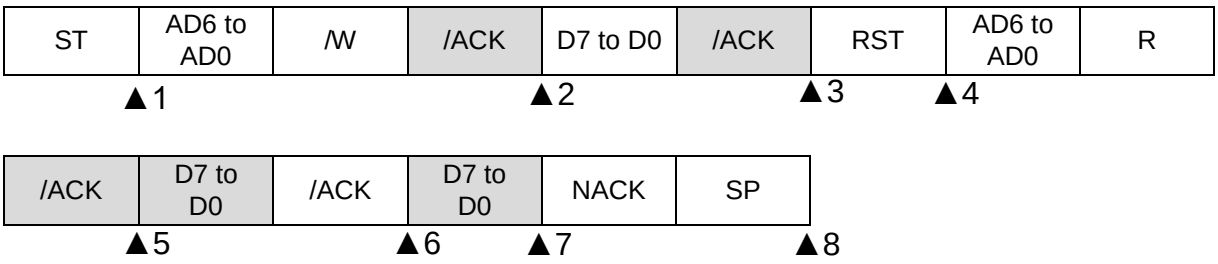
▲2: RXI interrupt: Address transmission completed (transfer direction bit: read)

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

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

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

5.2.3 Master Transmit/Receive



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

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

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

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

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

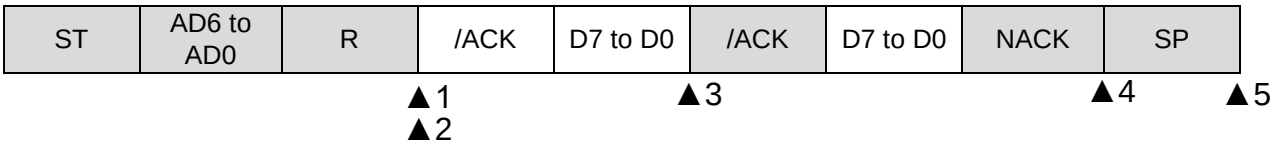
▲6: RXI interrupt: Reception for the last data - 1 completed (second data)

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

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

5.2.4 Slave Transmission

When transmitting 2-byte data:



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

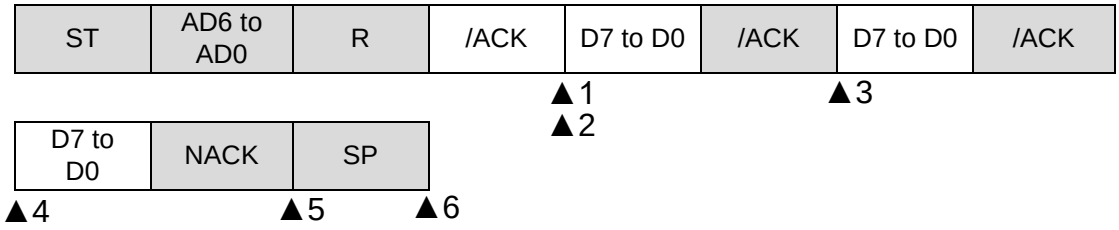
▲2: TXI interrupt: Transmit buffer is empty

▲3: TXI interrupt: Transmit buffer is empty

▲4: EEI (NACK) interrupt: NACK detected

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

When transmitting 3-byte data:



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

▲2: TXI interrupt: Transmit buffer is empty

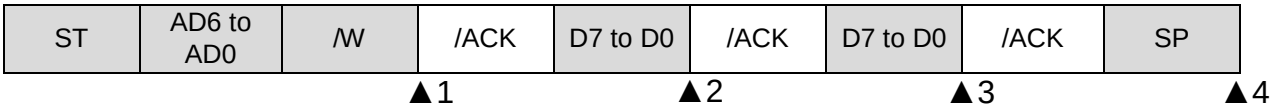
▲3: TXI interrupt: Transmit buffer is empty

▲4: TXI interrupt: Transmit buffer is empty

▲5: EEI (NACK) interrupt: NACK detected

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

5.2.5 Slave Reception



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

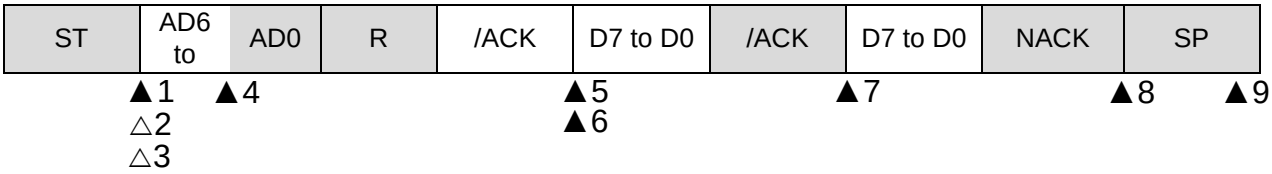
▲2: RXI interrupt: Reception for the last data - 1 completed (second data)

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

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

5.2.6 Multi-Master Communication

(Slave transmission after detecting AL during master transmission)



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

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

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

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

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

▲6: TXI interrupt: Transmit buffer is empty

▲7: TXI interrupt: Transmit buffer is empty

▲8: EEI (NACK) interrupt: NACK detected

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

5.3 Timeout Detection and Processing After the Detection

5.3.1 Detecting a Timeout with the Timeout Detection Function

When the timeout detection function is enabled by the setting in r_riic_config.h, call the R_RIIC_GetStatus() function in the callback function.

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

- When the TMO bit is 1: Timeout detected

- When the TMO bit is 0: Timeout not detected

5.3.2 Processing After a Timeout is Detected

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

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

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

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

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

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

Figure 5.5 shows the Timeout Detection and Processing After the Detection.

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

If the RX111 Group is used, refer to “27.11.2 Extra SCL Clock Cycle Output Function” in the RX111 Group User’s Manual: Hardware.

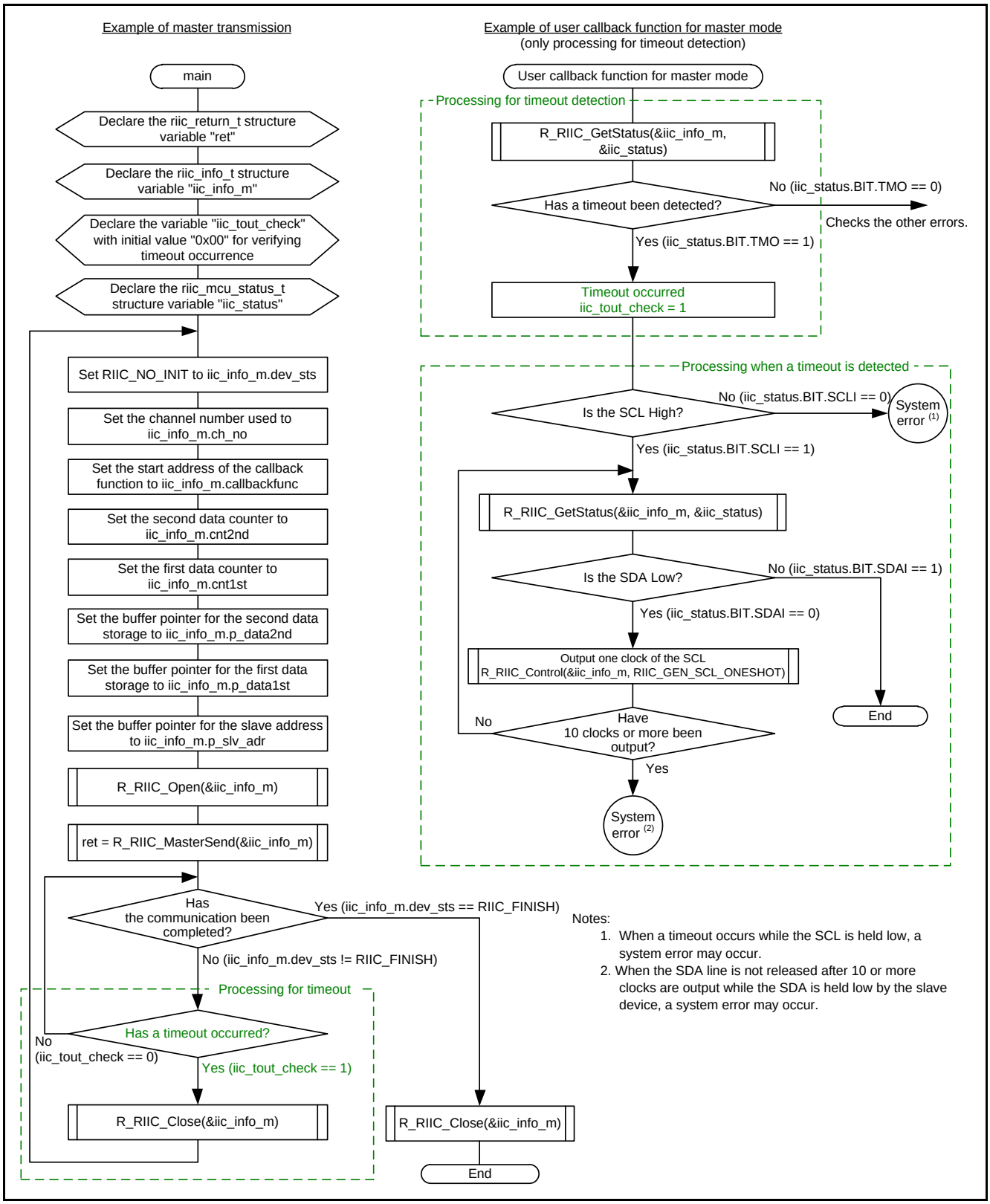


Figure 5.5 Timeout Detection and Processing After the Detection

5.4 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 V3.1.0.024
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.01.01
	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 RX231 (product number. R0K505231SxxxBE) 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. RTK5005130SxxxxxBE) 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 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)

Table 5-10 Operation Confirmation Environment for Rev.2.10.

Item	Contents
Integrated deveropment environment	Renesas Electronics e² studio V5.3.0.023
C compiler	Renesas Electronics C/C++ compiler for RX Family V.2.06.00
	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.2.10
Board used	Renesas Starter Kit for RX24T (product number. RTK500524TSxxxxxBE) Renesas Starter Kit for RX24U (product number. RTK500524USxxxxxBE)

Table 5-11 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 RX130-512KB (product number. RTK5051308SxxxxxBE) Renesas Starter Kit+ for RX65N-2MB (product number. RTK50565N2SxxxxxBE)

5.5 Troubleshooting

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

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

- When using CS+:

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

- When using e² studio:

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

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

(2) Q: I have added the FIT module to the project and built it. Then I got the error: This MCU is not supported by the current r_riic_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_riic_rx_config.h" may be wrong. Check the file "r_riic_rx_config.h". If there is a wrong setting, set the correct value for that. Refer to 2.7 Configuration Overview for details.

5.6 Sample Code

5.6.1 Example when Accessing One Slave Device Continuously with One Channel

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

The procedure is as follows:

1. Execute the R_RIIC_Open function to use RIIC channel 0 in the RIIC FIT module.
2. Execute the R_RIIC_MasterSend function to write 16-byte data to EEPROM.
3. Performs Acknowledge Polling to wait for EEPROM write completion.
4. Execute the R_RIIC_MasterReceive function to write 16-byte data from EEPROM.
5. Compare write data with read data.
6. Execute the R_RIIC_Close function to release RIIC channel 0 from the RIIC FIT module.

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

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

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

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

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

/* variables */
static volatile riic_return_t ret;   /* Return value */
static riic_info_t iic_info_m;       /* Structure data */

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

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

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

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

Figure 5.6 Example when Accessing One Slave Device Continuously with One Channel (1/5)

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

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

    /* Set arguments for R_RIIC_Open. */
    iic_info_m.ch_no = 0;               /* Channel number */
    iic_info_m.dev_sts = RIIC_NO_INIT; /* Device state flag (to be updated) */

    ret = R_RIIC_Open(&iic_info_m);
    if (RIIC_SUCCESS != ret)
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIIC_SUCCESS'. */
        while (1)
        {
            nop();    /* error */
        }
    }

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

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

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

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

    ret = R_RIIC_Close(&iic_info_m);
    if (RIIC_SUCCESS != ret)
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIIC_SUCCESS'. */
        while (1)
        {
            nop();    /* error */
        }
    }

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

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

Figure 5.7 Example when Accessing One Slave Device Continuously with One Channel (2/5)

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

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

} /* End of function callback_master() */

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

    /* Master send start */
    ret = R_RIIC_MasterSend(&iic_info_m);
    if (RIIC_SUCCESS == ret)
    {
        /* Waitting for R_RIIC_MasterSend completed. */
        while (RIIC_COMMUNICATION == iic_info_m.dev_sts)
        {
            /* do nothing */
        }

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

Figure 5.8 Example when Accessing One Slave Device Continuously with One Channel (3/5)

```
        }
    }

} /* End of function eeprom_write() */


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

        /* Master send start. */
        ret = R_RIIC_MasterSend(&iic_info_m);
        if (RIIC_SUCCESS == ret)
        {
            /* Waitting for R_RIIC_MasterSend completed. */
            while (RIIC_COMMUNICATION == iic_info_m.dev_sts)
            {
                /* do nothing */
            }

            /* Slave returns NACK. Set retry interval. */
            if (RIIC_NACK == iic_info_m.dev_sts)
            {
                /* Waitting for retry interval 100us. */
                R_BSP_SoftwareDelay(100, BSP_DELAY_MICROSECS);
            }
        }
        else
        {
            /* This software is for single master.
               Therefore, return value should be always 'RIIC_SUCCESS'. */
            while (1)
            {
                nop();    /* error */
            }
        }
    } while (RIIC_FINISH != iic_info_m.dev_sts);

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

Figure 5.9 Example when Accessing One Slave Device Continuously with One Channel (4/5)

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

    /* Master send receive start. */
    ret = R_RIIC_MasterReceive(&iic_info_m);
    if (RIIC_SUCCESS == ret)
    {
        /* Waitting for R_RIIC_MasterSend completed. */
        while (RIIC_COMMUNICATION == iic_info_m.dev_sts)
        {
            /* do nothing */
        }

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

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

Figure 5.10 Example when Accessing One Slave Device Continuously with One Channel (5/5)

6. Reference Documents

User's Manual: Hardware

The latest version can be downloaded from the Renesas Electronics website.

Technical Update/Technical News

The latest information can be downloaded from the Renesas Electronics website.

User's Manual: Development Tools

RX Family Compiler CC-RX User's Manual (R20UT3248)

The latest versions can be downloaded from the Renesas Electronics website.

Related Technical Updates

This module reflects the content of the following technical updates.

- TN-RX*-A012A/E

Website and Support

Renesas Electronics website

http://www.renesas.com

Inquiries

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

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Aug. 1, 2013	-	First edition issued
1.10	Sep. 30, 2013	-	Modified return values.
1.20	Nov. 15, 2013	4	Limitations: Changed the interrupt size to 120 bytes in (6).
		5	Table 1.2 Required Memory Size: - Changed the Size for the ROM to 7340 bytes. - Changed the Size for the Maximum interrupt stack usage to 120 bytes.
		47	Figure 4.2 State Transition on Master Transmission (R_RIIC_MasterSend() Called): - Added an arrow to indicate EV7 from STS8 to STS2. - Modified the comment on the arrow from STS8 to STS1.
		48	Figure 4.3 State Transition on Master Reception (R_RIIC_MasterReceive() Called): - Added an arrow to indicate EV7 from STS8 to STS2. - Modified the comment on the arrow from STS8 to STS1.
1.30	Apr. 1, 2014	-	Added support for the RX100 Series.
1.40	Oct. 1, 2014	1	Target Device: Changed from the RX100 Series to the RX111, RX110 and RX64M Groups.
		1	Related Documents: Added.
		4	1. Overview: - Features supported by this module: Added the description regarding channel 0 of RX64M in the third item. - Limitations: - Added the DMAC to (1) as the module not supported with this module. - Deleted (2), (5) and (6) in rev.1.30. - Added (5) to (7).
		5	Table 1.2 Required Memory Size: Changed the memory sizes.
		18	Figure 1.14 RIIC FIT Module State Transition Diagram: Added “RIIC_TMO” in the Error state.
		19	Table 1.2 Device State Flags when Transitioning States: Added “Timeout detection state”.
		20	1.3.8 Timeout Detection Function: Added.
		21	2.2 Software Requirements: Deleted “r_cgc_rx”.
		22 to 26	2.6 Configuration Overview: - Added parameters for CH2. - Changed the explanation of the following parameters: RIIC_CFG_CH0_kBPS, RIIC_CFG_CH0_SCL0, RIIC_CFG_CH0_SDA0 - Deleted the parameter “RIIC_CFG_PCLK_Hz”. - Deleted the parameter “RIIC_CFG_CH0_INT_PRIORITY” and added separated parameters for RXI, TXI, EEI, and TEI (e.g. RIIC_CFG_CH0_RXI_INT_PRIORITY). - Added parameters regarding timeout detection. - Added note 1.
		27	2.7 Parameters: Added the description regarding the limitation of rewriting the structure.
		27	2.8 Return Values: Added “RIIC_ERR_TMO”.
		29	3.1 R_RIIC_Open(): Added the limitation of rewriting the structure to the explanation in the Parameters.

Rev.	Date	Description	
		Page	**Summary**
1.40	Oct. 1, 2014	31 to 39	3.2 R_RIIC_MasterSend(),3.3 R_RIIC_MasterReceive(), and 3.4 R_RIIC_SlaveTransfer(): - Parameters: Added the limitation of rewriting the structure to the explanation. - Return Values: Added “RIIC_ERR_TMO”. - Example: Changed the code in the CallbackMaster function. - Special Notes (3.4 only): Changed description in the Notes.
		40, 41	3.5 R_RIIC_GetStatus(): - Changed the structure members of “riic_mcu_status_t”. - Changed the flag allocation table in the Special Notes.
		42	3.6 R_RIIC_Control(): - Parameters: Added the limitation of rewriting the structure to the explanation. - Special Notes: Added “One-shot output of the SCL clock”.
		44	3.7 R_RIIC_Close(): Added the limitation of rewriting the structure to the explanation in the Parameters.
		47 to 60	4. Appendices: Changed symbols for interrupt names “ICEEI”, “ICTEI”, “ICRXI” and “ICTXI” to “EEI”, “TEI”, “RXI” and “TXI”, respectively.
		47	Table 4.1 States Used for Protocol Control: Added state STS10 “RIIC_STS_TMO”.
		47	Table 4.2 Events Used for Protocol Control: - Added EV10 “RIIC_EV_INT_TMO”.
		49, 50	Figure 4.2 State Transition on Master Transmission and Figure 4.3 State Transition on Master Reception: - Added descriptions regarding state STS10 (RIIC_STS_TMO). - Deleted the arrow from STS8 to STS9.
		51	Figure 4.4 State Transition on Slave Transmission and Reception: Deleted descriptions regarding STS9 (RIIC_STS_AL).
		52	Table 4.3 Protocol State Transition Table: - Added the column for EV10 and the row for STS10. - Changed “FuncA” to “Func10”.
		53	Table 4.4 Functions Used on Protocol State Transition: - Changed “FuncA” to “Func10”. - Added the row for Func11 “riic_time_out()”.
		54	Table 4.5 States of Flags on State Transitions: - Added “RIIC_STS_TMO” for all the “Communicating” states. - Deleted “RIIC_STS_AL” from the “Communicating (slave transmission/reception” states. - Added the row for “Timeout detection state”.
		55 to 58	4.2 Interrupt Request Generation Timing: - Deleted notes 1 and 2.
		57	4.2.4 Slave Transmission: - When transmitting 2-byte data: Added “5: EEI (STOP) interrupt”. - When transmitting 3-byte data: Added “4: TXI interrupt”.
		58	4.2.6 Multi-Master Communication: Added.
		59, 60	4.3 Timeout Detection and Processing After the Detection: Added including Figure 4.5. .
		61	6. Reference Documents: Changed reference documents in the User’s Manual: Development Tools.

		Description	
Rev.	**Date**	**Page**	**Summary**
1.40	Oct. 1, 2014	Program	The module is updated to fix the software issue. Description: Slave communication is not available after an arbitration-lost occurs, and then the bus is locked. Conditions: The issue occurs when the following four conditions are all met. - RIIC FIT module rev. 1.30 or earlier is used. - RX device operates as both the master and the slave in multi-master communication. - An arbitration-lost is detected when communicating as the master. - Communication other than master reception or slave reception is performed. Measure: Please use the RIIC FIT module Rev. 1.40.
1.50	Dec. 1, 2014	-	Added support for the RX113 Group.
1.60	Dec. 15, 2014	-	Added support for the RX71M Group.
1.70	Dec. 15, 2014	-	Added support for the RX231 Group.
1.80	Oct. 31, 2015	-	Added support for the RX130 Group, RX230 Group, RX23T Group.
		34	Example of 3.2, R_RIIC_MasterSend(), modified
		37, 38	Example of 3.3, R_RIIC_MasterReceive(), modifided
		40, 41	Example of 3.4, R_RIIC_SlaveTransfer(), modified
1.90	Mar. 4, 2016	-	Added support for the RX24T Group.
		5	Table 1.2 Required Memory Size, changed.
		22, 28	Added description of r_riic_rx_pin_config.h to section 2.6, Configuration Overview.
		-	Changed “master composite” to “master transmit/receive”.
2.00	Oct 1, 2016	-	Added support for the RX65N Group.
		29	Changed code size description from “Table 1.2 Required Memory Size” to “2.7 Code Size.”
		Program	Corrected an error of the definitions “RIIC_IR_RXI2” and “RIIC_IR_TXI2” to refer the RXI, and TXI Interrupt Status Flag of channel 2.
		Program	The module is updated to fix the software issue. Description: Since there is an error in the handling of pin function settings of RX110 in Rev.1.90, build error occurs if use RX110. Conditions: When you build the project, after create a new project with selected "RX110" series device as MCU, and added RIIC FIT module Rev.1.90 in reference to "2.10 Adding the FIT Module to Your Project". Corrective action: Corrected the handling pin function settings by function riic_mcu_mpc_enable() and riic_mcu_mpc_disable(). Please use the RIIC FIT module Rev.2.00.
2.10	Jun 2, 2017	-	Added RX24U Group in the Target Device.
		-	Added support for the RX24T-512KB version.
		22	2.4. Usage of Interrupt Vector: Added.
		32	2.11. Callback Functions: Added.
		32	2.12. Adding the FIT Module to Your Project: Changed.
		52	4. Pin Settings: Added.
		69 to 70	5.4. Operating Test Environment: Added.

Rev.	Date	Description	
		Page	Summary
2.10	Jun 2, 2017	72	5.5. Troubleshooting: Added.
2.20	Aug. 31, 2017	-	Added support for the RX65N-2MB version.
		-	Added support for the RX130-512KB version.
		1	Related Documents: Added the following document: “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)”
		22	2.4. Usage of Interrupt Vector: Revised. Interrupt vector used in RX65N-2MB added to the Table 2.1. Interrupt Vector used in the RIIC FIT Module.
		24	2.7. Configuration Overview: Changed the description for RIIC_CFG_PORT_SET_PROCESSING.
		24 to 27	2.7. Configuration Overview: Added definitions for Channel 1.
		32	2.12. Adding the FIT Module to Your Project: Revised.
		52	4. Pin Settings: Revised.
		70	Table 5.11. Operation Test Environment for Rev.2.20, added.
		72 to 76	5.6. Sample Code: Added.
		77	6. Provided Modules: Deleted.
		Program	Added definitions for Channel 1.

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

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

1. Handling of Unused Pins

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

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

2. Processing at Power-on

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

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

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

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

4. Clock Signals

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

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

5. Differences between Products

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

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

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other disputes involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawing, chart, program, algorithm, application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics products.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (space and undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. When using the Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat radiation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions or failure or accident arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please ensure to implement safety measures to guard them against the possibility of bodily injury, injury or damage caused by fire, and social damage in the event of failure or malfunction of Renesas Electronics products, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures by your own responsibility as warranty for your products/system. Because the evaluation of microcomputer software alone is very difficult and not practical, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please investigate applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive carefully and sufficiently and use Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall not use Renesas Electronics products or technologies for (1) any purpose relating to the development, design, manufacture, use, stockpiling, etc., of weapons of mass destruction, such as nuclear weapons, chemical weapons, or biological weapons, or missiles (including unmanned aerial vehicles (UAVs)) for delivering such weapons, (2) any purpose relating to the development, design, manufacture, or use of conventional weapons, or (3) any other purpose of disturbing international peace and security, and you shall not sell, export, lease, transfer, or release Renesas Electronics products or technologies to any third party whether directly or indirectly with knowledge or reason to know that the third party or any other party will engage in the activities described above. When exporting, selling, transferring, etc., Renesas Electronics products or technologies, you shall comply with any applicable export control laws and regulations promulgated and administered by the governments of the countries asserting jurisdiction over the parties or transactions.
10. Please acknowledge and agree that you shall bear all the losses and damages which are incurred from the misuse or violation of the terms and conditions described in this document, including this notice, and hold Renesas Electronics harmless, if such misuse or violation results from your resale or making Renesas Electronics products available any third party.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

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

(Rev.3.0-1 November 2016)

RENESAS

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

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

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

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

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

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

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

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

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

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

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

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

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

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

Colophon 6.0