RX600, RX200 Series

I²C Bus Single Master Control Software Using RIIC Serial Interface

R01AN1254EJ0103
Rev.1.03
Jan. 30, 2015

Introduction

This application note describes I²C bus single master control using the RX Family I²C bus interface RIIC (RIIC), sample code that implements that control, and use of the sample code.

In this application note, the software used to control the slave device is referred to as the upper layer and the software that implements I²C single master basic protocol control as the lower layer. Slave devices are controlled by combining the protocols provided by the upper and lower layers.

This sample code implements the lower layer used for I²C single master control. The user should acquire or implement software corresponding to the upper level for slave device control.

Note that Renesas provides sample software for controlling slave devices under separate cover. This sample software is available if required.

Target Devices

Microcontroller: RX62N, RX63N, RX63T, RX210, RX21A

Device used for verifying operation: Renesas Electronics Corporation R1EX24xxx Series I²C Serial EEPROM.

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

Contents

1. Specifications

This sample code performs I^{2}C bus single master control using the RX Family I^{2}C bus interface. The user should acquire or implement software corresponding to the upper layer for slave device control.

Table 1.2 lists the used peripheral functions and their uses and figure 1.1 shows a usage example.

The following provides an overview of the functions provided by this software.

- This sample code is an I^{2}C bus single master device driver that uses the RX Family microcontroller as the master device using its I^{2}C bus interface.
- This sample code implements the protocols in the I^{2}C-bus specification. It supports master transmission, master reception, and master composite (master transmission → master reception) operation.
- Four transmission patterns can be set up for master transmission. Table 1.1 lists the operating patterns.
- The sample code supports multiple channels. Simultaneous communication using multiple channels is possible.
- Multiple slave devices with different type name can be controlled on a channel bus. However, while communication is in progress (the period from when the start condition occurs to when the stop condition occurs), communication with other devices is not possible.
- Communication is implemented by functions (start functions) that start various protocol control operations and the function (the advance function) that monitors communication and advances the processing. The communication state can be determined from the return values from the advance function.
- The start functions generate the start condition. The operations following that until the stop condition is generated are performed by calling the advance function to perform the processing forward.
- Interrupts are generated on completion of start condition generation, slave address transmission, data transmission, data reception, and stop condition generation.
- The communication rate can be set by the user. (Supported rates: up to 400 kHz (max)) However, if multiple devices are connected on the same channel, the communication rate must be set to match that of the slowest device.
- If communication is stopped by the influence of noise or other issues (in cases where an interrupt is not generated), an error can be returned from the advance function. If the number of advance function calls exceeds the limit, the sample code determines that communication has stopped due to an abnormal situation and a “no response error” is returned. This upper limit can be set by the user.
- If a NACK error occurs, a stop condition is occurred.
- The sample code provides SCL clock generation processing. If a synchronization discrepancy occurs between the master and slave due to noise or other problem and the I^{2}C bus goes to the SDA = low hold state, the SCL pseudo clock generation function can be called to force the slave device internal state to successful and terminate.
- This sample code only supports communication between 7-bit address devices. Special addresses (e.g. general call addresses) are not supported.

Table 1.1 Master Communication Operation Patterns

	ST Generation	Slave Address Transmission	First Data Transmission	Second Data Transmission	SP Generation
Pattern 1	○	○	○	○	○
Pattern 2	○	○	—	○	○
Pattern 3	○	○	—	—	○
Pattern 4	○	—	—	—	○

Legend:
ST: Start condition
SP: Stop condition

Table 1.2 Peripheral Function and Its Application

Peripheral Function	Application
RIIC	I^{2}C bus interface One channel (required)

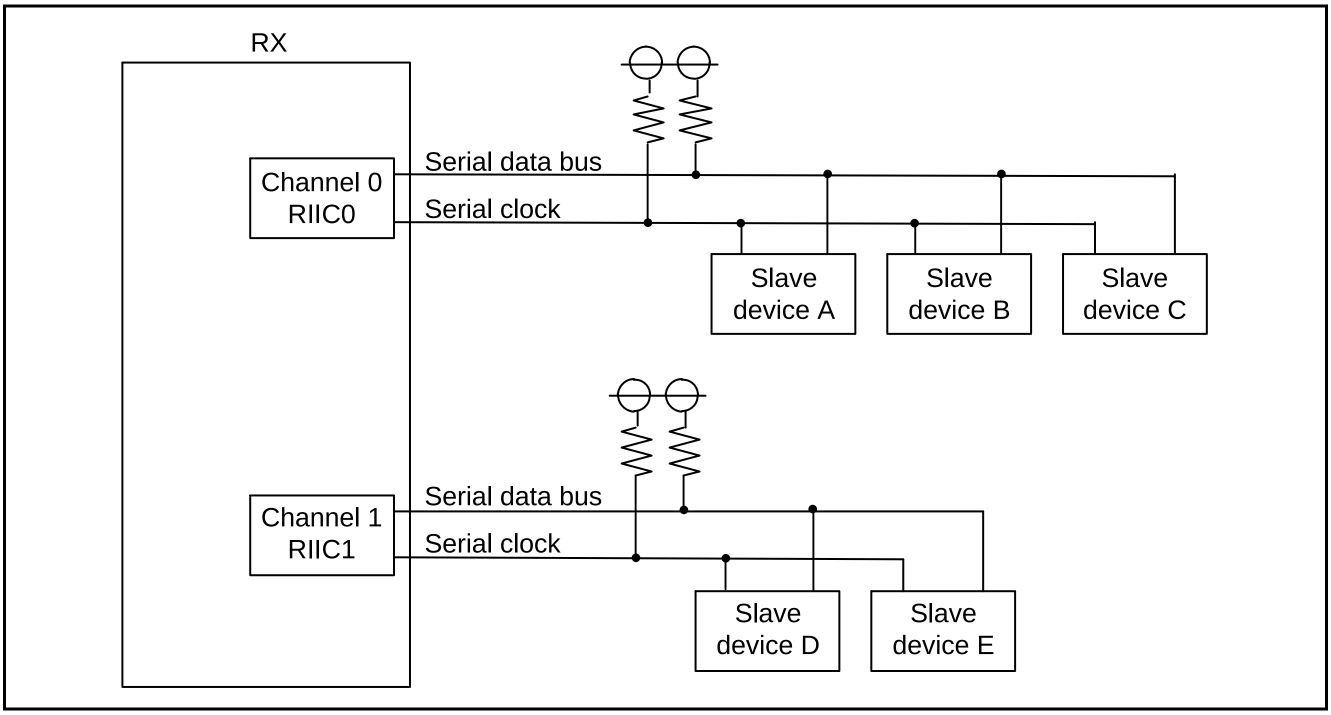


Figure 1.1 Usage Example

2. Operation Confirmation Conditions

The sample code accompanying this application note has been run and confirmed under the conditions below.

(1) RX62N

Table 2.1 Operation Confirmation Conditions

Item	Contents
MCU used for evaluation	RX62N Group (program ROM: 512 KB, RAM: 64 KB)
Memory used for evaluation	Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I^{2}C Serial EEPROM
Operating frequency	ICLK: 96 MHz, PCLK: 48 MHz
Operating voltage	3.3 V
Integrated development environment	Renesas Electronics High-performance embedded Workshop Version 4.09.01.007
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 1.2.1.0)
	Compile option Default settings*1 of integrated development environment used as compile options. Note: 1. Optimization level: 2; optimization method: optimize for size
Endian mode	Big endian/Little endian
Sample code version	Ver. 1.13
Software used	Renesas Electronics Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01
Board used	Renesas Starter Kit for RX62N

(2) RX63N

Table 2.2 Operation Confirmation Conditions

Item	Contents
MCU used for evaluation	RX63N Group (program ROM: 1 MB, RAM: 128 KB)
Memory used for evaluation	Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I^{2}C Serial EEPROM
Operating frequency	ICLK: 96 MHz, PCLKB: 48 MHz
Operating voltage	3.3 V
Integrated development environment	Renesas Electronics High-performance embedded Workshop Version 4.09.01.007
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 1.2.1.0)
	Compile option Default settings*1 of integrated development environment used as compile options. Note: 1. Optimization level: 2; optimization method: optimize for size
Endian mode	Big endian/Little endian
Sample code version	Ver. 1.13
Software used	Renesas Electronics Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01
Board used	Renesas Starter Kit for RX63N

(3) RX63T

Table 2.3 Operation Confirmation Conditions

Item	Contents
MCU used for evaluation	RX63T Group (program ROM: 512 KB, RAM: 64 KB)
Memory used for evaluation	Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I²C Serial EEPROM
Operating frequency	ICLK: 96 MHz, PCLKB: 48 MHz
Operating voltage	3.3 V
Integrated development environment	Renesas Electronics CubeSuite+ V2.00.00
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 2.00.00)
	Compile option Default settings*[1] of integrated development environment used as compile options. Note: 1. Optimization level: 2; optimization method: optimize for size
Endian mode	Big endian/Little endian
Sample code version	Ver. 1.13
Software used	Renesas Electronics Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01
Board used	Renesas Starter Kit for RX63T

(4) RX210

Table 2.4 Operation Confirmation Conditions

Item	Contents
MCU used for evaluation	RX210 Group (program ROM: 512 KB, RAM: 64 KB)
Memory used for evaluation	Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I²C Serial EEPROM
Operating frequency	ICLK: 50 MHz, PCLKB: 25 MHz
Operating voltage	3.3 V
Integrated development environment	Renesas Electronics High-performance embedded Workshop Version 4.09.01.007
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 1.2.1.0)
	Compile option Default settings*[1] of integrated development environment used as compile options. Note: 1. Optimization level: 2; optimization method: optimize for size
Endian mode	Big endian/Little endian
Sample code version	Ver. 1.13
Software used	Renesas Electronics Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01
Board used	Renesas Starter Kit for RX210

(5) RX21A

Table 2.5 Operation Confirmation Conditions

Item	Contents
MCU used for evaluation	RX21A Group (program ROM: 512 KB, RAM: 64 KB)
Memory used for evaluation	Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I^{2}C Serial EEPROM
Operating frequency	ICLK: 50 MHz, PCLKB: 25 MHz
Operating voltage	3.3 V
Integrated development environment	Renesas Electronics High-performance embedded Workshop Version 4.09.01.007
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 1.2.1.0)
	Compile option Default settings*[1] of integrated development environment used as compile options. Note: 1. Optimization level: 2; optimization method: optimize for size
Endian mode	Big endian/Little endian
Sample code version	Ver. 1.13
Software used	Renesas Electronics Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01
Board used	HSBRX21AP-B (Hokuto Denshi Co., Ltd.)

3. Reference Application Note

For additional information associated with this document, refer to the following application note.

- Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ)

4. Peripheral Functions

The RX Family microcontrollers provide two I^{2}C bus control peripheral functions: the I^{2}C bus interface and serial communication interface simplified I^{2}C bus module.

This application note uses the I^{2}C bus interface.

5. Hardware

5.1 Pins Used

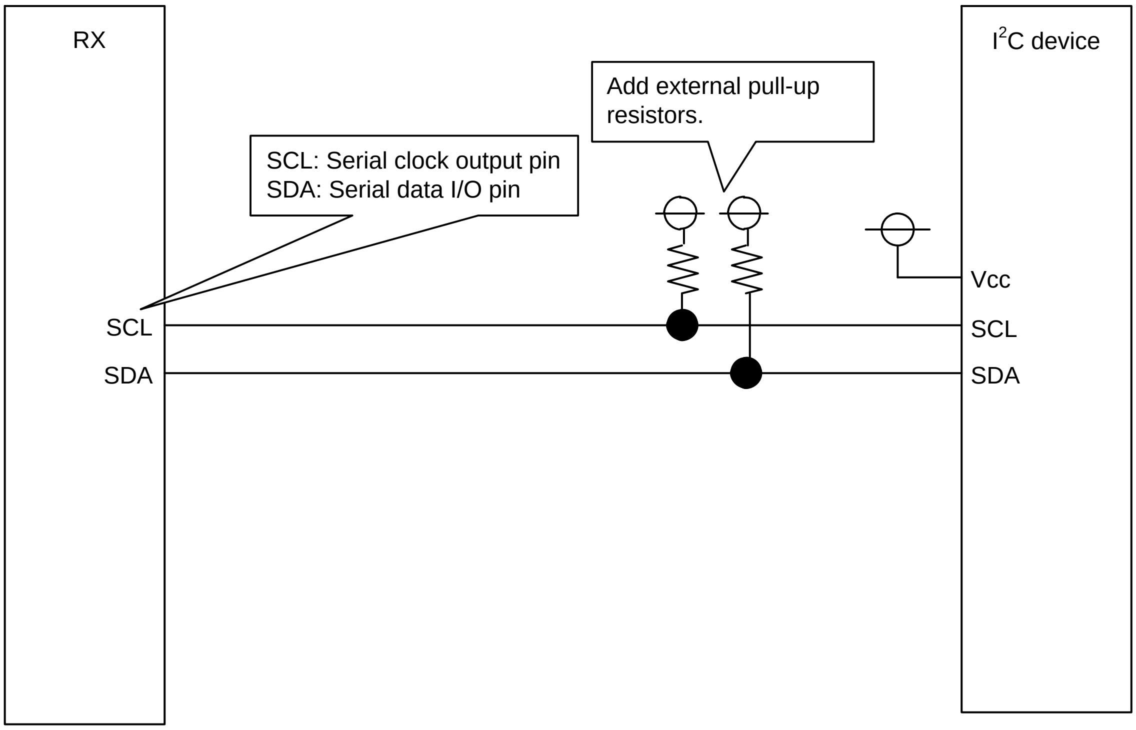
Table 5.1 lists the Pins Used and Their Functions.

Table 5.1 Pins Used and Their Functions

Pin Name	I/O	Description
SCL (SCL in figure 5.1)	Output	Serial clock output
SDA (SDA in figure 5.1)	I/O	Serial data I/O

5.2 Reference Circuit

Figure 5.1 shows an example connection between the RX Family I^{2}C bus interface and an I^{2}C slave device. Since the output is N-ch open drain, the serial clock line and serial data bus line require external pull-up resistors. Select resistors that are appropriate for the system. Also consider adding damping resistors to the signal lines to ensure matching circuit characteristics.



- The pins on the MCU used for serial I/O depend on the MCU model.
- These pins are designated as the SCL pin and SDA pin in this application note to match the notations used in the sample code.

Figure 5.1 Connection Between RX Family I^{2}C Bus Interface and I^{2}C Slave Device

5.3 Controlling Multiple Slave Devices

The sample code supports use of multiple channels. In addition, multiple slave devices with different type name can be connected to a channel bus and controlled. However, communication with other devices is not possible during the period from when the start condition occurs to when the stop condition occurs.

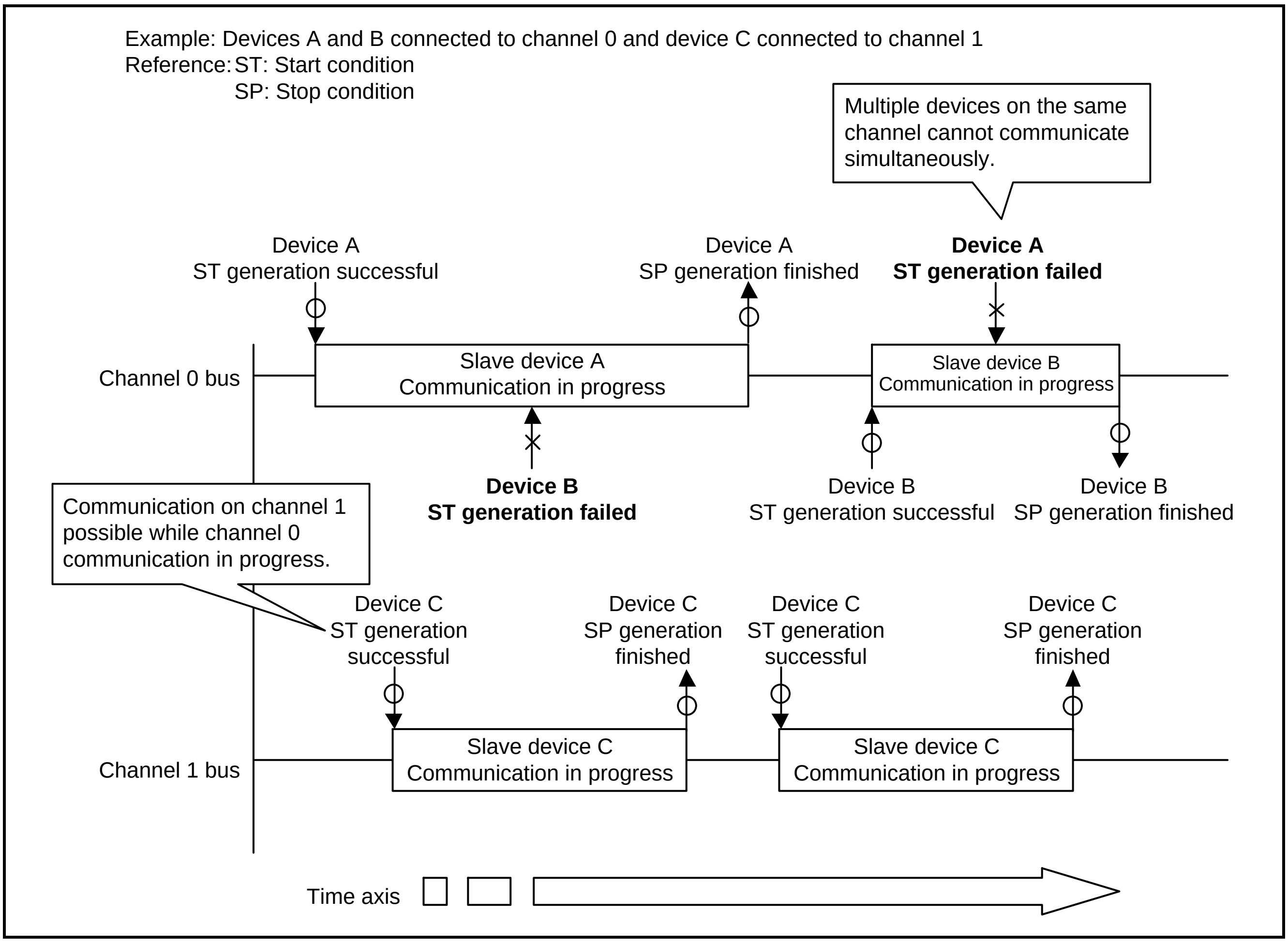


Figure 5.2 Example of Control of Multiple Slave Devices

5.4 Maximum Transfer Speed

The maximum transfer speed setting is 400 kHz.

However, when both standard mode and fast mode devices are connected to the same channel, the standard mode maximum setting of 100 kHz must be observed.

The maximum transfer speeds of mixed bus systems are listed below.

Table 5.2 Maximum Transfer Speeds of Mixed Bus Systems

Communication Device	Mixed Devices	
	Fast Mode	Standard Mode
Fast mode	0 to 400 kHz	0 to 100 kHz
Standard mode	0 to 100 kHz	0 to 100 kHz

6. Software

6.1 Software Structure

This sample code takes the software used to control slave devices as the upper layer and the software that implements I^{2}C bus single master basic protocol control to be the lower layer. The upper layer combines protocols provided by the lower layer to control slave devices.

This sample code is positioned as lower layer used for I^{2}C bus single master control.

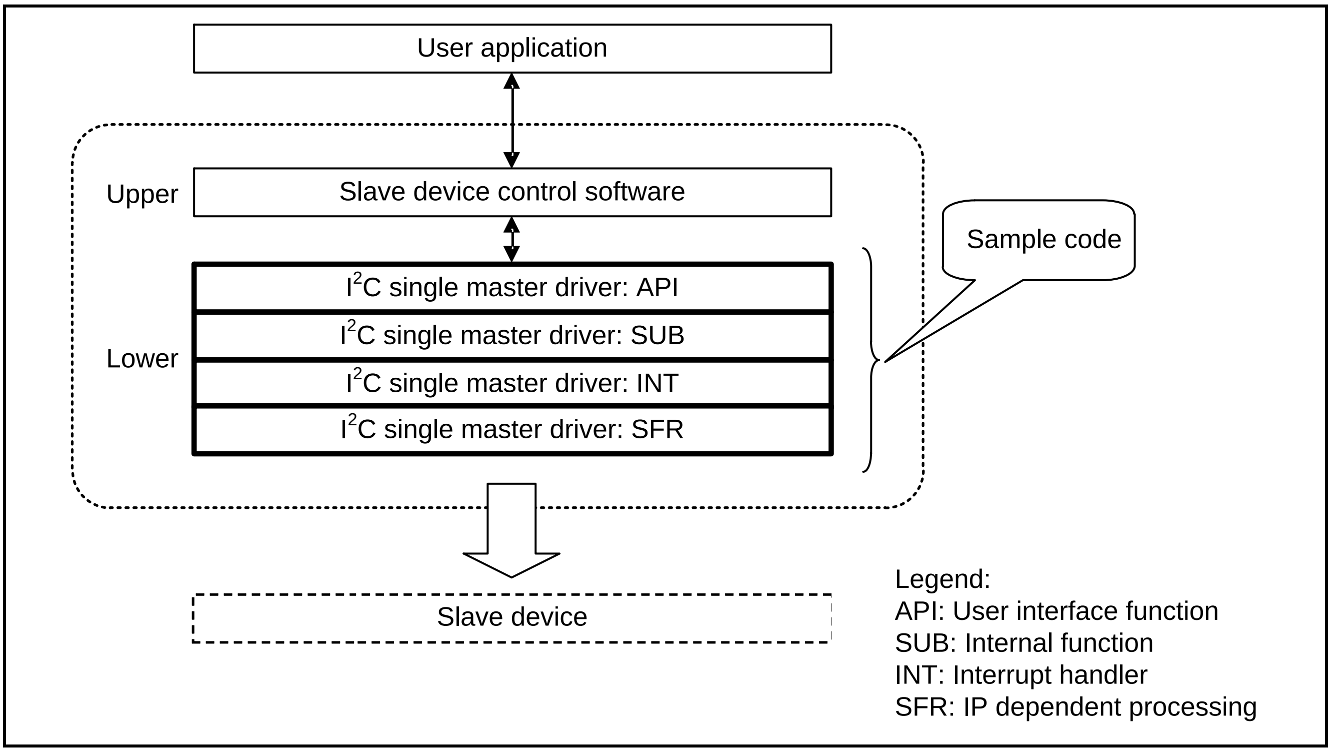


Figure 6.1 Software Structure

6.2 Operation Overview

This sample code implements I^{2}C bus single master control using the RX Family MCU I^{2}C bus interface.

In particular, it implements the following single master protocols.

Table 6.1 Control Protocols

No.	Control Protocol	Outline
1	Master transmission	Transfers data from the master (microcontroller) to the slave device. There are four transmission patterns that can be used.
2	Master reception	The master (microcontroller) receives data from the slave device.
3	Master composite	After master transmission, a master reception operation is performed.

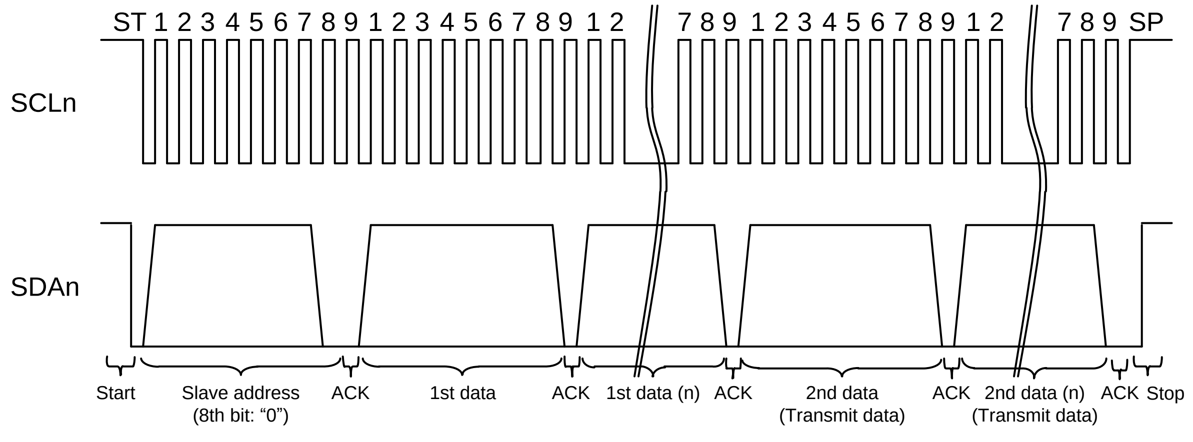
6.2.1 Master Transmission

There are four transmission patterns that can be used for master transmission. The function can be selected by the method used to set up the I^{2}C communication information structure, which manages the communication information. See section 6.13.1, Communication information structure, for details on setting up this structure.

(1) Pattern 1

Data is transferred from the master (microcontroller) to the slave device.

First, a start condition (ST) is generated and then the slave device address is transmitted. During this transmission, the 8th bit is the transfer direction specification bit and a 0 (write) is transmitted for data transmission. Next, 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 device 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 device. When a data transmission has been started and all data transmission has completed, a stop condition (SP) is generated, releasing the bus.



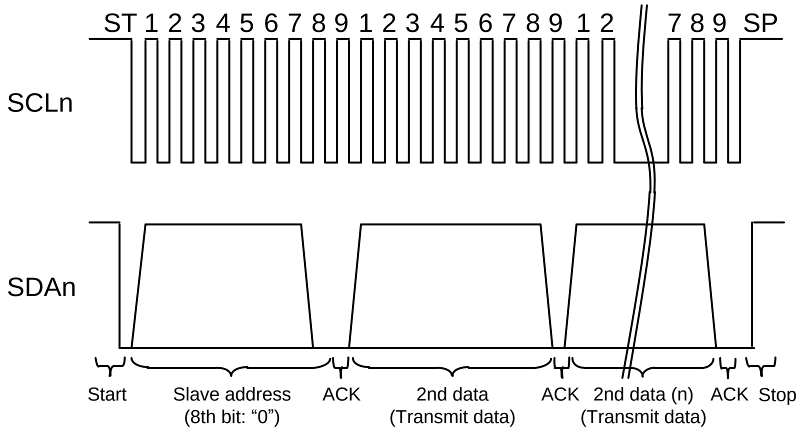
Legend:
ST: Start condition generation
SP: Stop condition generation
ACK: Acknowledge “0”

Figure 6.2 Master Transmission (Pattern 1) Signals

(2) Pattern 2

Data is transferred from the master (microcontroller) to the slave device. However, the first data is not transferred.

Operation from start condition (ST) generation through slave device address transmission is the same as for pattern 1. However, after that the second data is transferred without sending the first data. When all data transmission has completed, a stop condition (SP) is generated, releasing the bus.



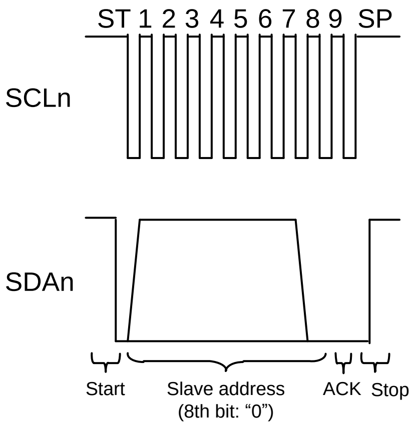
Legend:
ST: Start condition generation
SP: Stop condition generation
ACK: Acknowledge "0"

Figure 6.3 Master Transmission (Pattern 2) Signals

(3) Pattern 3

Operation from start condition (ST) generation through slave device address transmission is the same as in successful operation. In cases where neither the first data nor the second data are set up, however, a stop condition (SP) is generated releasing the bus without transferring any data.

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



Legend:
ST: Start condition generation
SP: Stop condition generation
ACK: Acknowledge "0"

Figure 6.4 Master Transmission (Pattern 3) Signals

(4) Pattern 4

In this pattern, after a start condition (ST) is generated, a stop condition (SP) is generated and released the bus without transmitting the slave address, first data, or second data when those data are not set up.

This pattern is useful for just releasing the bus.

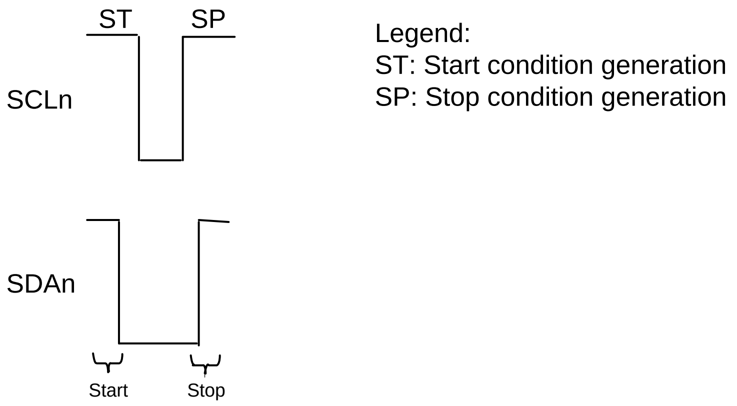


Figure 6.5 Master Transmission (Pattern 4) Signals

6.2.2 Master Reception

In master reception, the master (microcontroller) receives data from a slave device.

Here a start condition (ST) is generated and then the slave device address is transmitted. Since the 8th bit at this time is the transfer direction specification bit, a 1 (read) is transmitted when this data is transmitted. Next, data reception starts. Although an ACK is transmitted after each single byte of data is received during reception, a NACK is transmitted only after the last data to notify the slave device that reception processing has terminated. When all the data has been received, a stop condition (SP) is generated, releasing the bus.

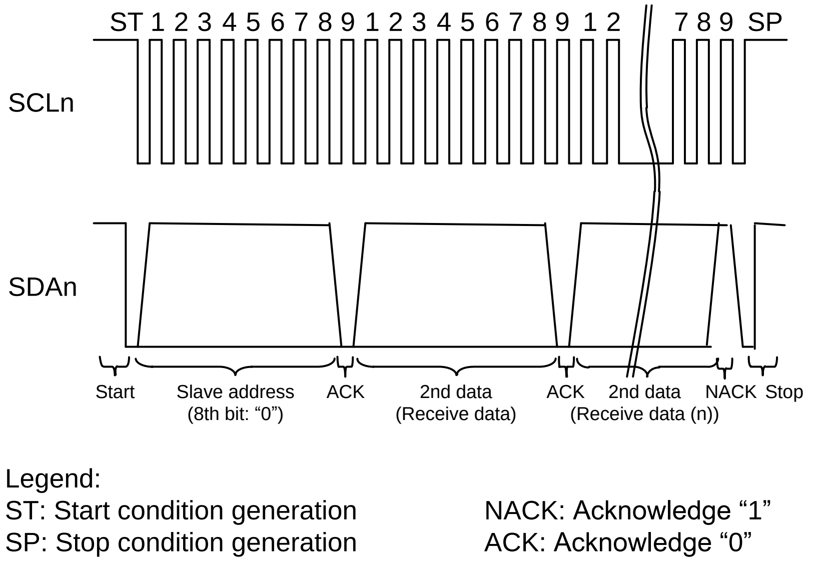


Figure 6.6 Master Reception Signals

6.2.3 Master Composite

In this mode, data is first transmitted from the master (microcontroller) to the slave device (master transmission). After this transmission completes, a restart condition is generated, the transfer direction is changed to 1 (read) and the master receives data from the slave device (master reception).

First, a start condition (ST) is generated and then the slave device address is transmitted. During this transmission, the 8th bit is the transfer direction specification bit and a 0 (write) is transmitted for data transmission. When the data transmission completes, a restart condition (RST) is generated and the slave address is transmitted. At this time, a 1 (read) is transmitted as the transfer direction specification bit. Next, data reception starts. Although an ACK is transmitted after each single byte of data is received during reception, a NACK is transmitted only after the last data to notify the slave device that reception processing has terminated. When all the data has been received, a stop condition (SP) is generated, releasing the bus.

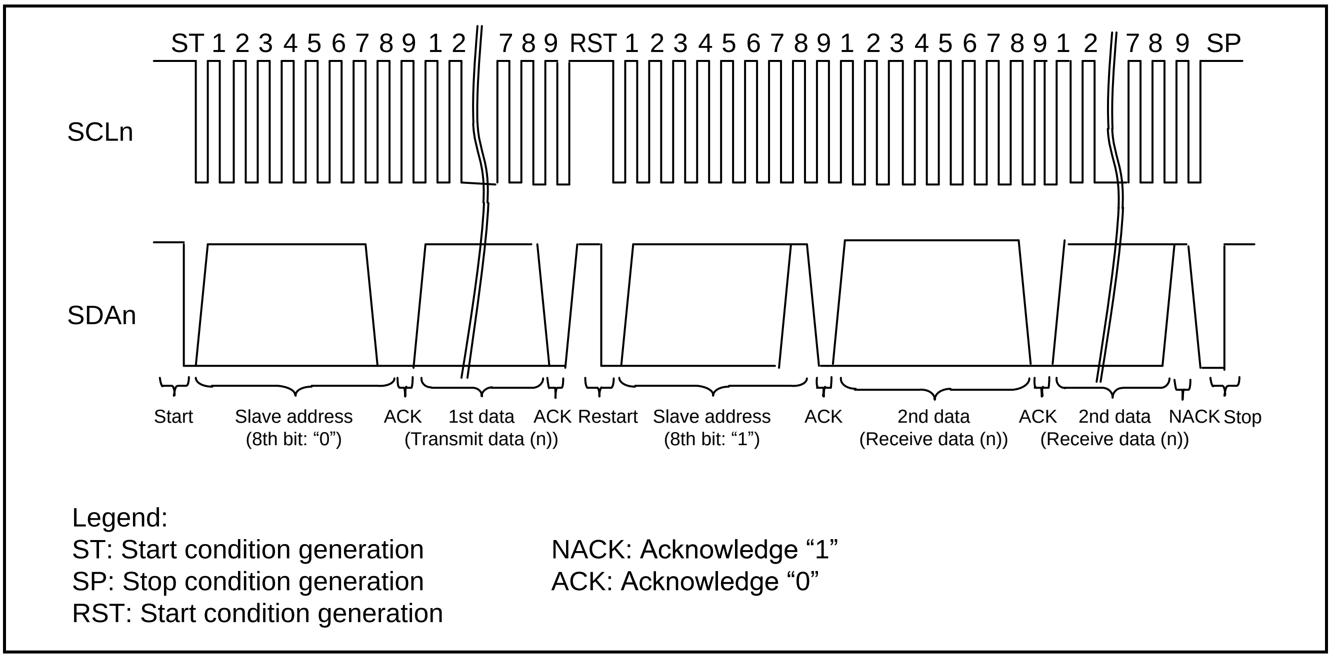


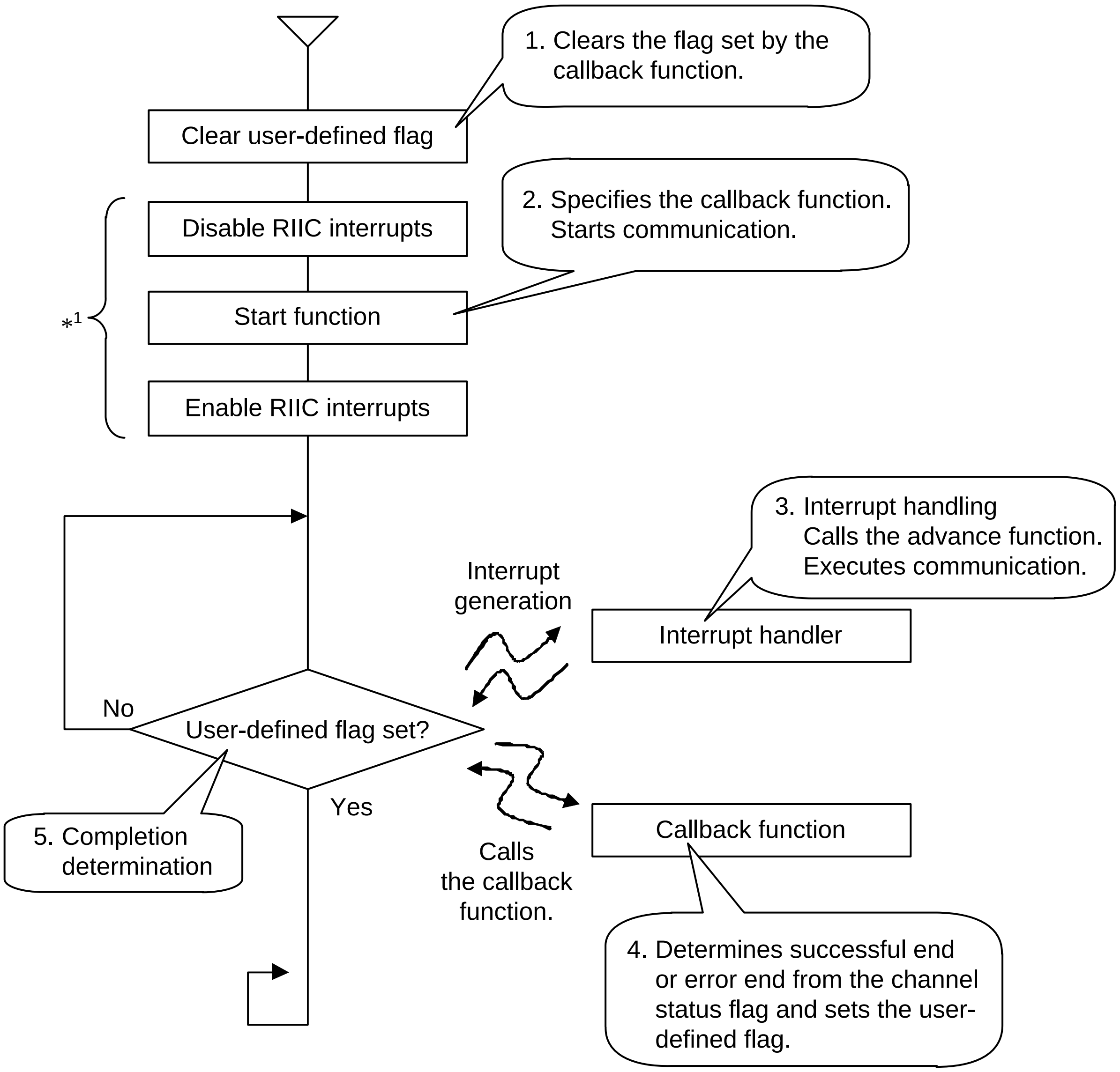
Figure 6.7 Master Composite Signals

6.3 Software Operation

Communication is started by calling the start function. After that, I^{2}C bus communication is moved forward by the user calling the advance function. Two modes of software operation, one in which the advance function is called by the RIIC interrupt handler and one in which it is called by the main processing routine, are described below.

(1) Calling the Advance Function from the RIIC Interrupt Handler

Communication is started by calling the start function. To confirm that communication has finished, specify a callback function to set a flag, etc. The callback function is called when either a successful end or an error end occurs. It is possible to determine whether communication ended successfully or with an error by reading the channel status flag (g_iic_ChStatus[]).



Note: 1. If the RIIC interrupt handler is used to call the advance function, processing should be added by the user for disabling and enabling RIIC interrupts before the start functions are called. If an RIIC interrupt occurs in the interval before a start function is called, multiple API calls will overlap and processing will end before the advance function can run. This will prevent subsequent communication from occurring.

Figure 6.8 Software Operation Example: Calling the Advance Function from the RIIC Interrupt Handler

(2) Calling the Advance Function from the Main Processing Routine

Communication is started by calling the start function. Continue calling the advance function from the main processing routine until communication finishes. While communication is in progress, the status can be verified by checking the return values from the advance function.

An event flag (g_iic_Event[]) is set when an interrupt occurs. The advance function monitors the event flag (g_iic_Event[]) and executes communication when it confirms that the event flag (g_iic_Event[]) has been set. For details of the event flag, see table 6.20.

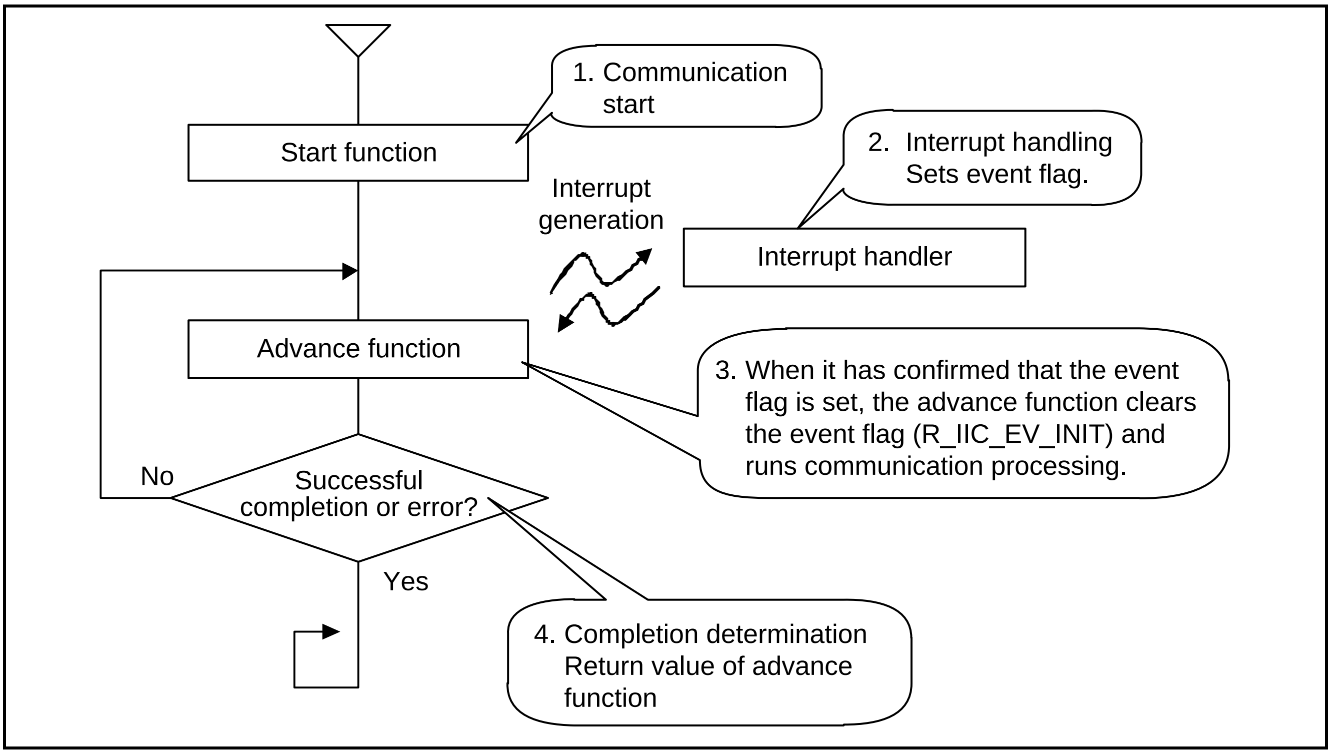


Figure 6.9 Software Operation Example: Calling the Advance Function from the Main Processing Routine

6.4 Software Operating Sequence

The figures below shows the operating sequence when the advance function is called by the RIIC interrupt handler and when it is called by the main processing routine.

(1) Calling the Advance Function from the RIIC Interrupt Handler

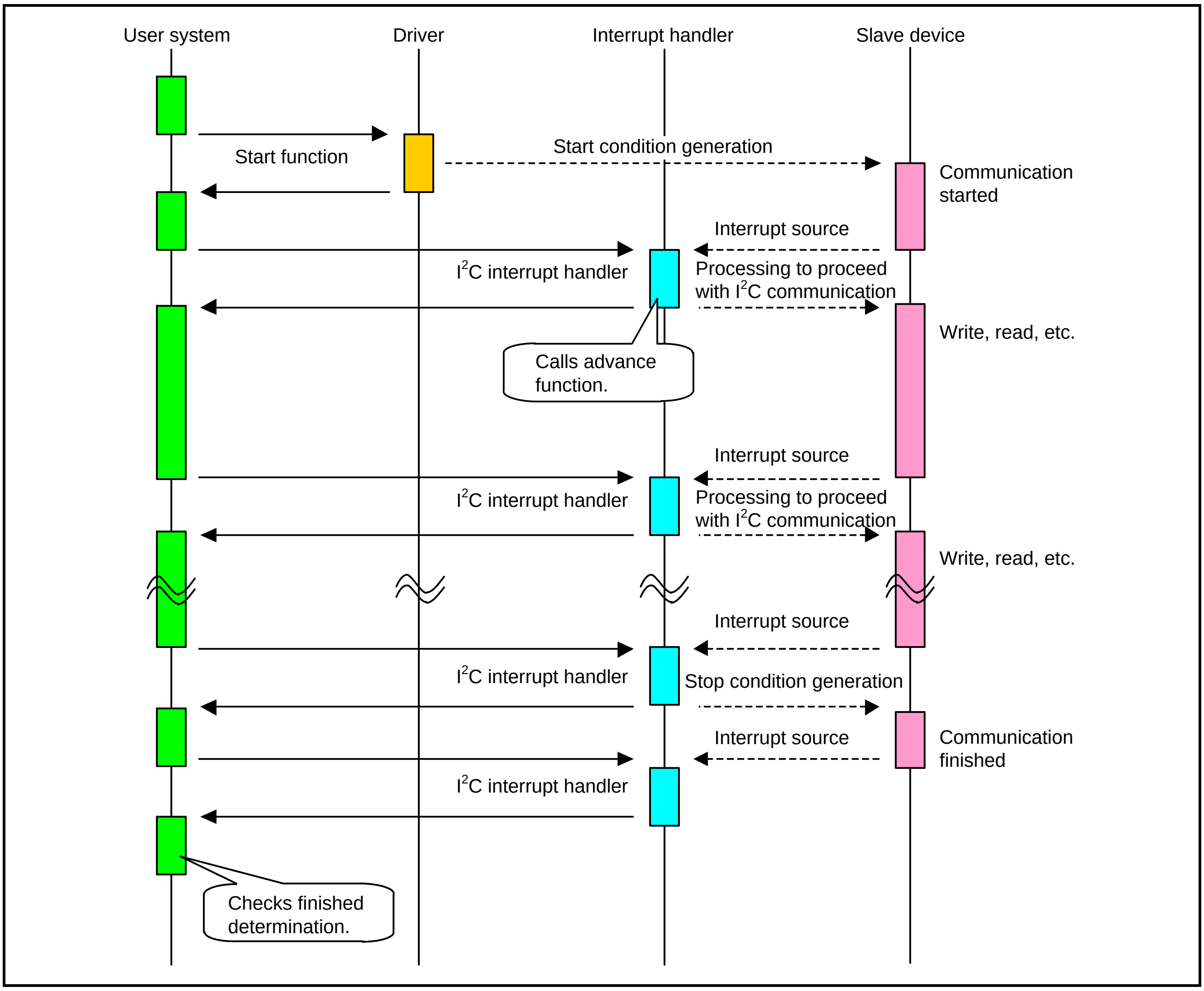


Figure 6.10 Sequence: Calling the Advance Function from the RIIC Interrupt Handler

(2) Calling the Advance Function from the Main Processing Routine

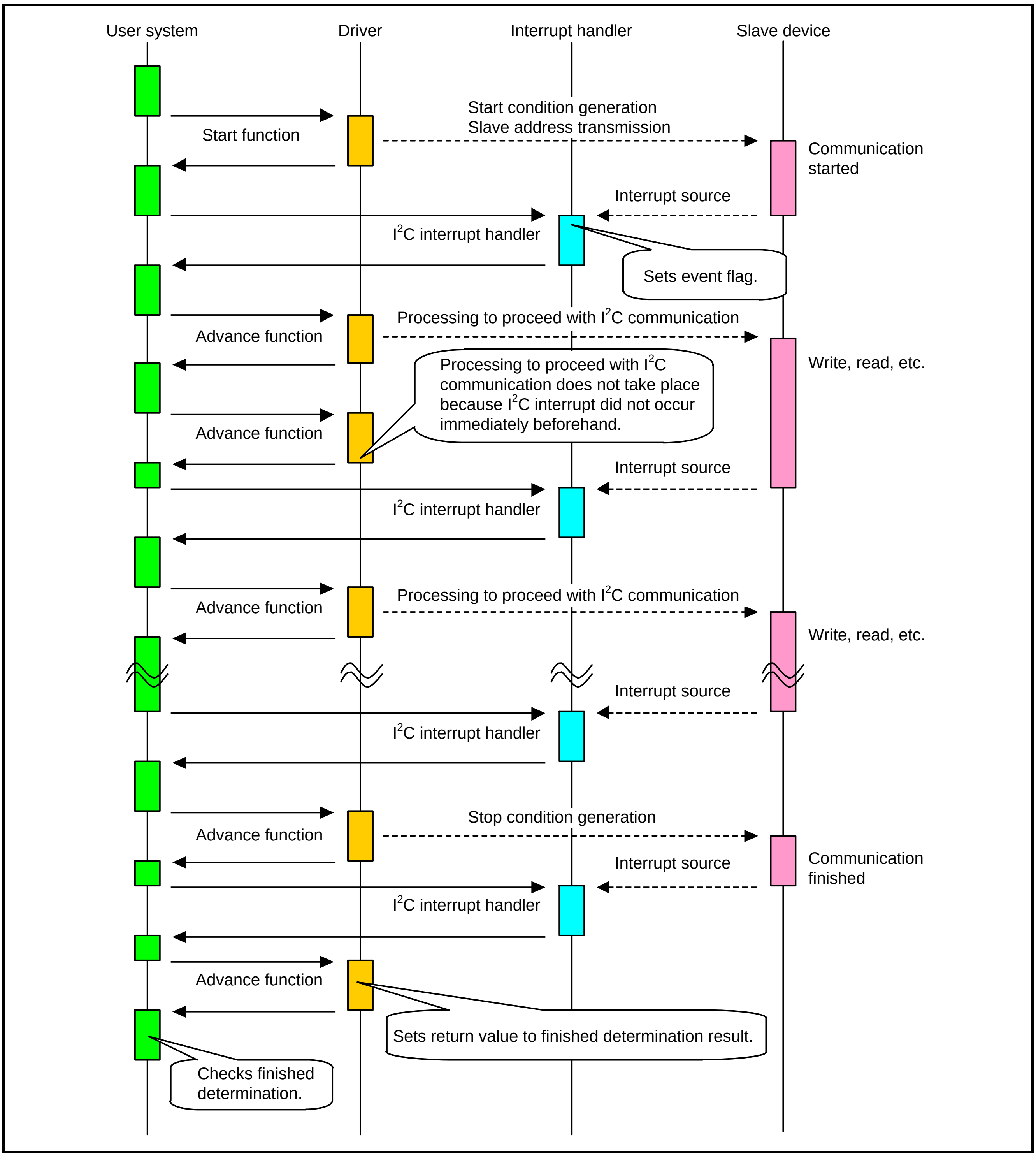


Figure 6.11 Sequence: Calling the Advance Function from the Main Processing Routine

6.5 Implementation of Slave Device Control

(1) Slave Device Management

Information such as the channels used and the communication data is managed in a structure. Communication between multiple devices on a channel is implemented by setting up a structure for each slave device controlled.

See section 6.13.1, I^{2}C Communication Information Structure for details on this structure.

(2) Channel Status Management

Exclusive control of multiple slave devices connected to a bus is implemented using the g_iic_ChStatus[] channel state flag. See the g_iic_ChStatus[] entry in section 6.15, Variables, for details on the channel state flags.

One of these flags exists for each channel and they are managed in a global variable. These flags are set to the R_IIC_IDLE/R_IIC_FINISH/R_IIC_NACK state (the idle state (communication possible)) if I^{2}C driver initialization completes and communication is not performed on the corresponding bus. The state of these flags is set to R_IIC_COMMUNICATION (communication in progress) during communication. Since these flags are always checked at the start of communication, communication with another device on the same channel will never be started during communication. Simultaneous communication over multiple channels is implemented by managing these flags for each channel.

(3) Device State Management

Control of multiple slave devices on the same channel is supported with the *pDevStatus device state flag member in the I^{2}C communication information structure. The communication state of the corresponding device is stored in the device state flag. See section 8.6, Control Methods for Multiple Slave Devices on the Same Channel, for details on the use of these flags.

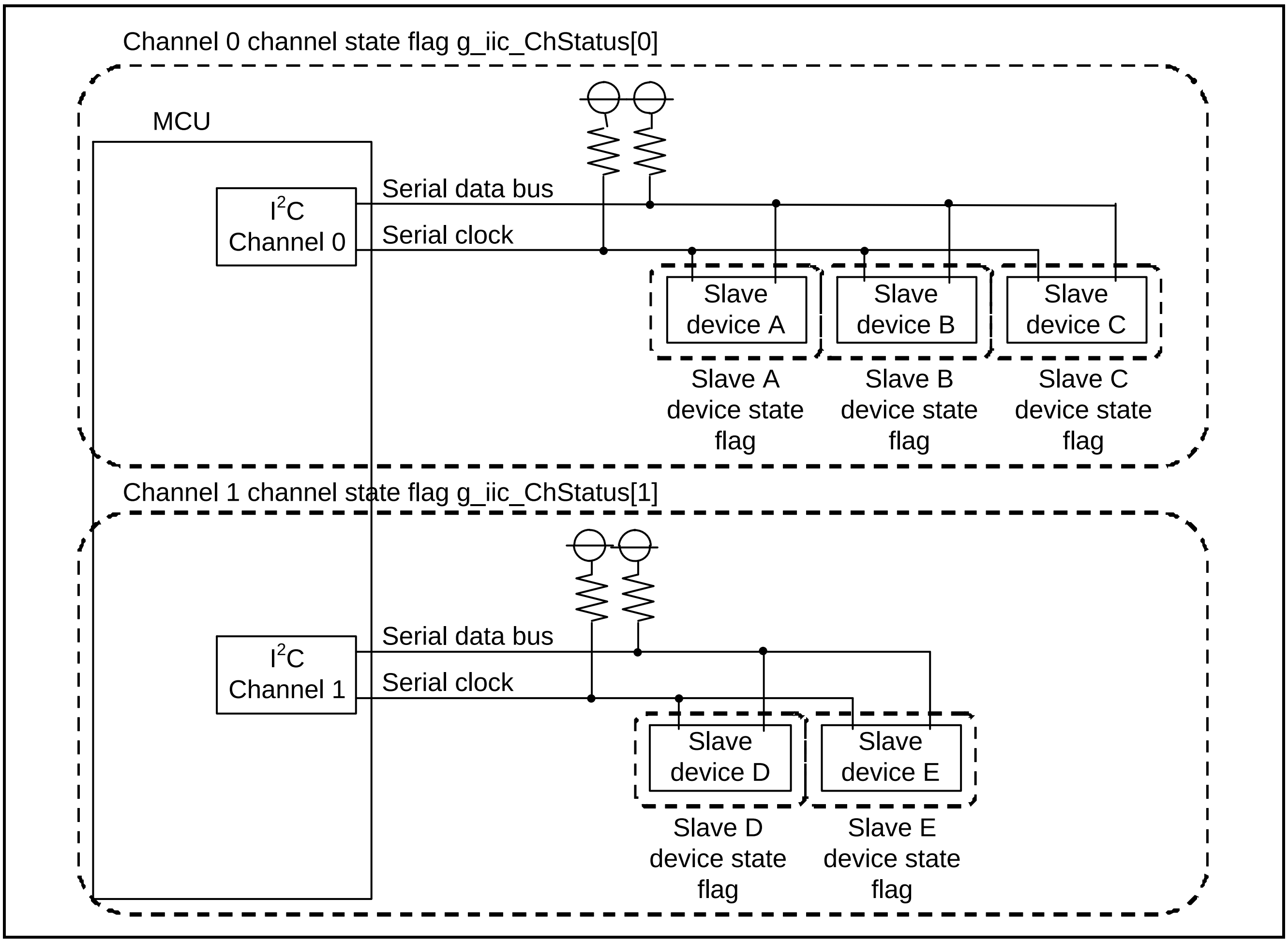


Figure 6.12 Slave Device Control

6.6 Communication Implementation

This sample code manages start conditions, slave device communication, and other processing as a single protocol, and implements communication combinations with this protocol.

6.6.1 States During Control

The following states are defined to implement protocol control.

Table 6.2 States Used for Protocol Control (enum r_iic_drv_internal_status_t)

No.	Constant Name	Description
STS0	R_IIC_STS_NO_INIT	Uninitialized state
STS1	R_IIC_STS_IDLE	Idle state
STS2	R_IIC_STS_ST_COND_WAIT	Start condition generation complete wait state
STS3	R_IIC_STS_SEND_SLVADR_W_WAIT	Slave address [Write] transmission complete wait state
STS4	R_IIC_STS_SEND_SLVADR_R_WAIT	Slave address [Read] transmission complete wait state
STS5	R_IIC_STS_SEND_DATA_WAIT	Data transmission complete wait state
STS6	R_IIC_STS_RECEIVE_DATA_WAIT	Data reception complete wait state
STS7	R_IIC_STS_SP_COND_WAIT	Stop condition generation complete wait state

6.6.2 Events During Control

The following events generated during protocol control are defined.

Note that not only interrupts, but calls the interface functions provided by this sample code are defined as events.

Table 6.3 Events Used for Protocol Control (enum r_iic_drv_internal_event_t)

No.	Event	Event Definition
EV0	R_IIC_EV_INIT	Call r_iic_drv_init_driver()
EV1	R_IIC_EV_GEN_START_COND	Call r_iic_drv_generate_start_cond()
EV2	R_IIC_EV_INT_START	ICEEI interrupt generation (interrupt flag: START)
EV3	R_IIC_EV_INT_ADD	ICTEI interrupt generation
EV4	R_IIC_EV_INT_SEND	ICTEI interrupt generation
EV5	R_IIC_EV_INT_RECEIVE	ICRXI interrupt generation
EV6	R_IIC_EV_INT_STOP	ICEEI interrupt generation (interrupt flag: STOP)
EV7	R_IIC_EV_INT_AL	ICEEI interrupt generation (interrupt flag: AL)
EV8	R_IIC_EV_INT_NACK	ICEEI interrupt generation (interrupt flag: NACK)

6.6.3 Protocol State Transitions

In this sample code, the state transitions on calls the provided interface functions and when I²C interrupts occur. The following figures show the protocol state transitions.

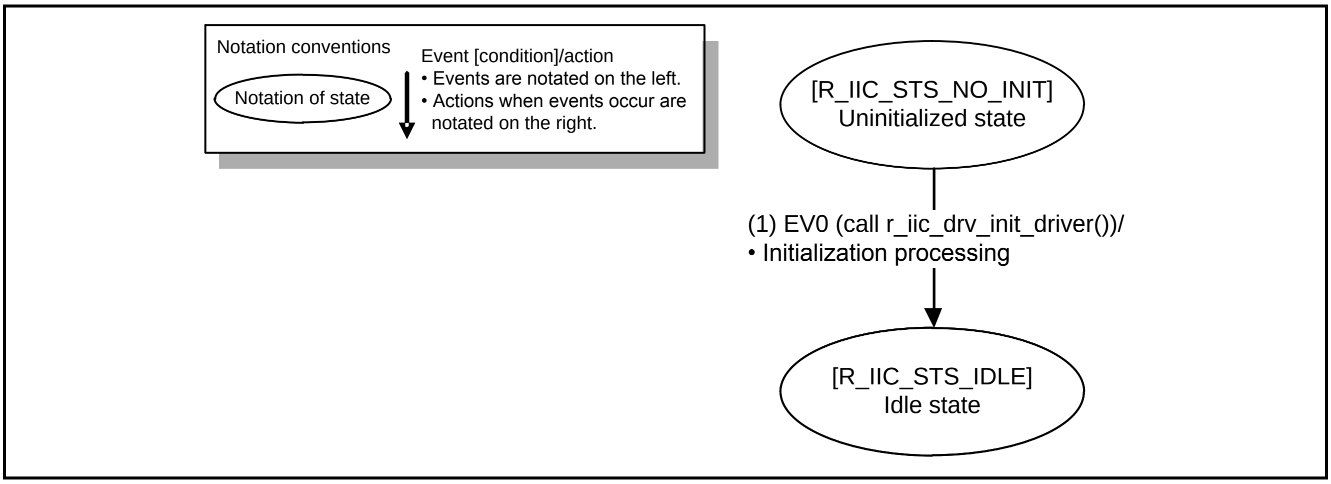


Figure 6.13 Initialization State Transition Diagram

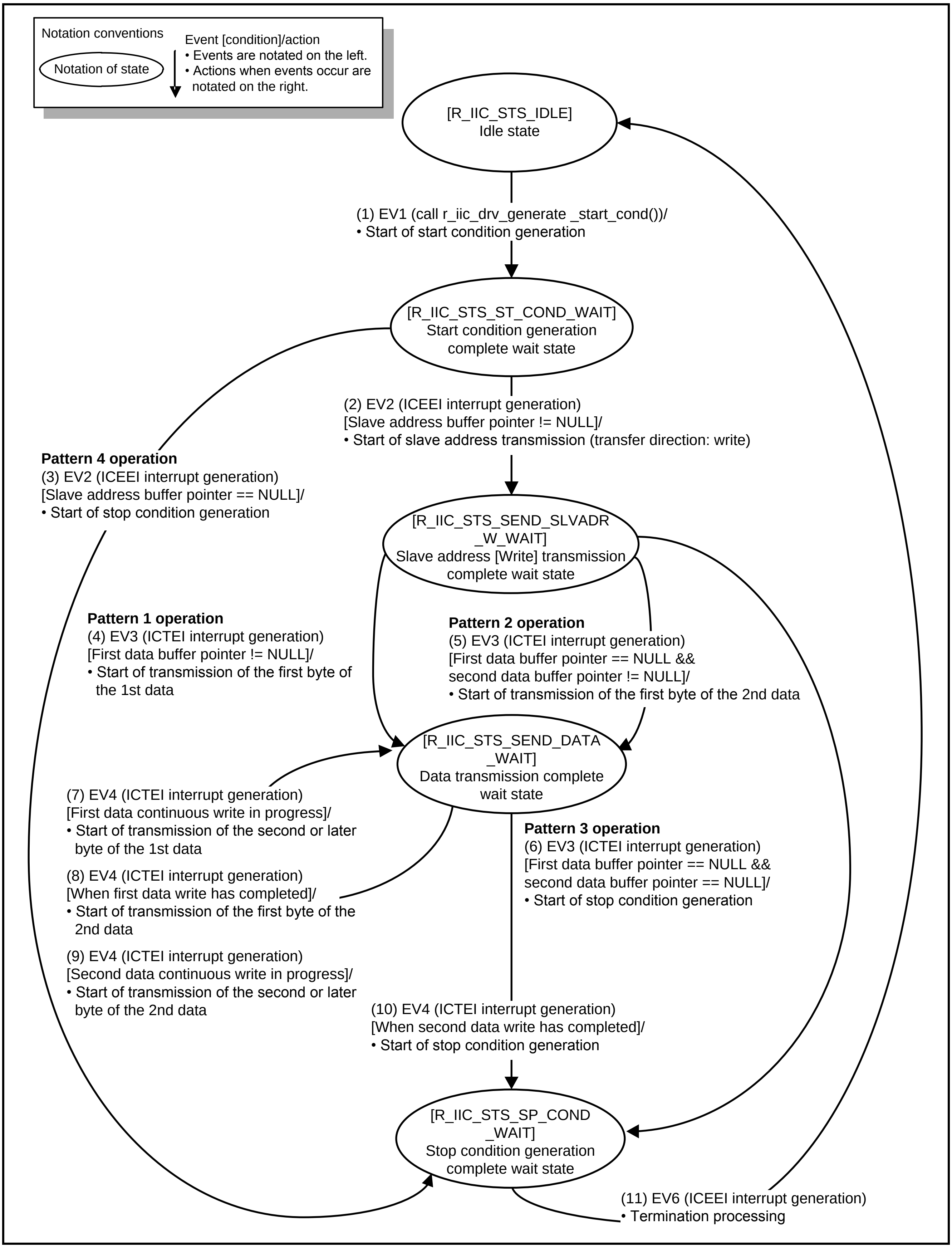


Figure 6.14 Master Transmission State Transition Diagram

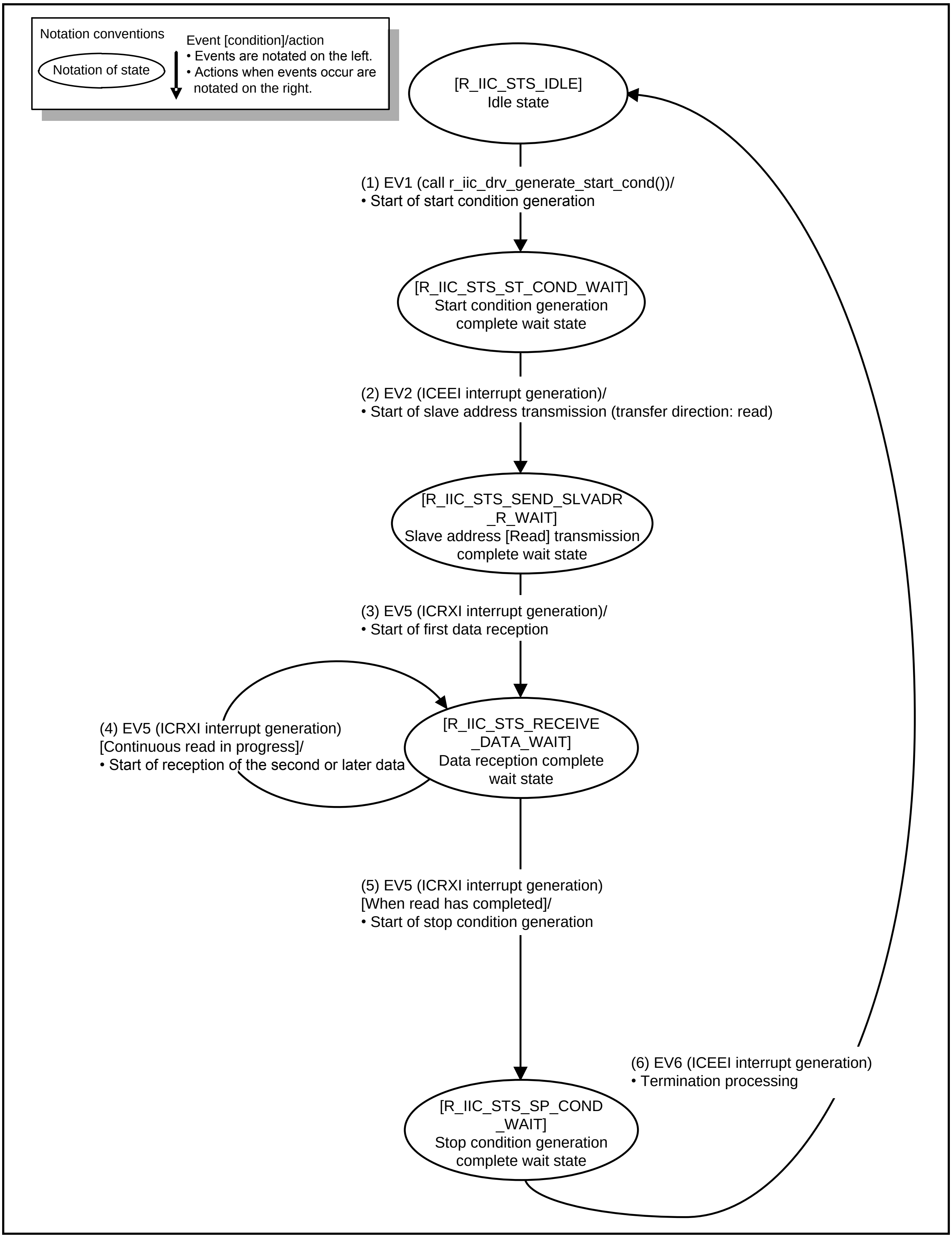


Figure 6.15 Master Reception State Transition Diagram

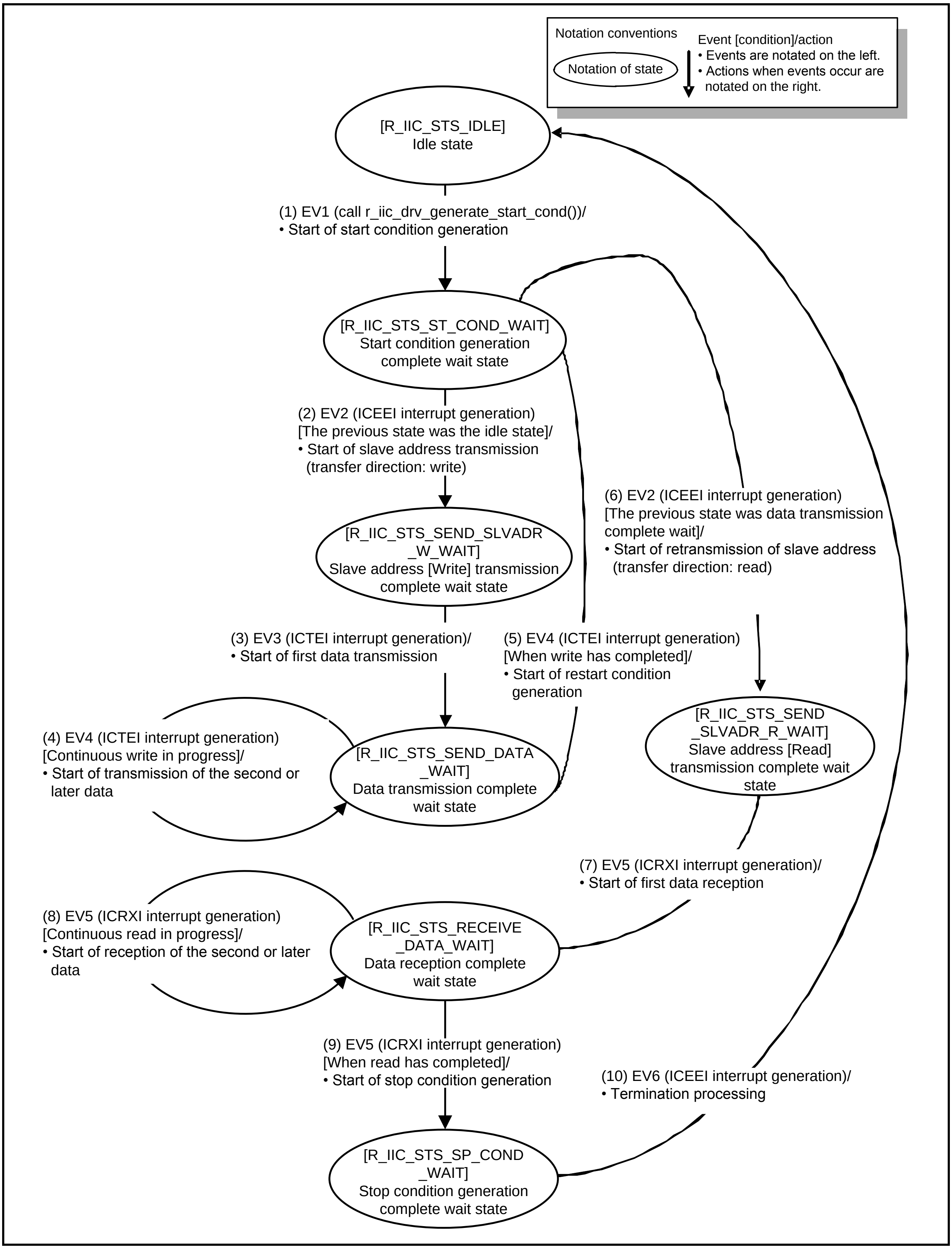


Figure 6.16 Master Composite State Transition Diagram

6.6.4 Protocol State Transition Table

The processing for the operations when the events in table 6.3 occur in the states shown in table 6.2 is defined in the following state transition table.

For STS0 and following states, see the "No." column in table 6.2. For EV0 and other events, see the "No." column in table 6.3. See table 6.5 for Func0 and the following functions.

Table 6.4 Protocol State Transition Table (gc_iic_mtx_tbl[][])

State		Event								
		EV0	EV1	EV2	EV3	EV4	EV5	EV6	EV7	EV8
STS0	Uninitialized state [R_IIC_STS_NO_INIT]	Func0	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
STS1	Idle state [R_IIC_STS_IDLE]	ERR	Func1	ERR	ERR	ERR	ERR	ERR	ERR	ERR
STS2	Start condition generation complete wait state [R_IIC_STS_ST_COND_WAIT]	ERR	ERR	Func2	ERR	ERR	ERR	ERR	Func7	Func8
STS3	Slave address [Write] transmission complete wait state [R_IIC_STS_SEND_SLVADR_W_WAIT]	ERR	ERR	ERR	Func3	ERR	ERR	ERR	Func7	Func8
STS4	Slave address [Read] transmission complete wait state [R_IIC_STS_SEND_SLVADR_R_WAIT]	ERR	ERR	ERR	ERR	ERR	Func3	ERR	Func7	Func8
STS5	Data transmission complete wait state [R_IIC_STS_SEND_DATA_WAIT]	ERR	ERR	ERR	ERR	Func4	ERR	ERR	Func7	Func8
STS6	Data reception complete wait state [R_IIC_STS_RECEIVE_DATA_WAIT]	ERR	ERR	ERR	ERR	ERR	Func5	ERR	Func7	Func8
STS7	Stop condition generation complete wait state [R_IIC_STS_SP_COND_WAIT]	ERR	ERR	ERR	ERR	ERR	ERR	Func6	Func7	Func8

Note: "ERR" indicates R_IIC_ERR_OTHER. Cases where an event that has no meaning in that state is reported are all handled as errors.

6.6.5 Protocol State Transition Registered Functions

The functions registered in the state transition table are defined as follows.

Table 6.5 Protocol State Transition Registered Functions

Processing	Function	Overview
Func0	r_iic_drv_init_driver()	Initialization
Func1	r_iic_drv_generate_start_cond()	Start condition generation
Func2	r_iic_drv_arter_gen_start_cond()	Processing after start condition generation
Func3	r_iic_drv_after_send_slvadr()	Post slave address transmission completion processing
Func4	r_iic_drv_write_data_sending()	Data transmission
Func5	r_iic_drv_read_data_receiving()	Data reception
Func6	r_iic_drv_release()	Communication termination
Func7	r_iic_drv_arbitration_lost()	Arbitration lost error handling
Func8	r_iic_drv_nack()	NACK error handling

6.6.6 Processing at Protocol State Transitions

This section describes the processing performed by r_iic_drv_func_table() (referred to below as the processing that advances communication) when a protocol state transition occurs.

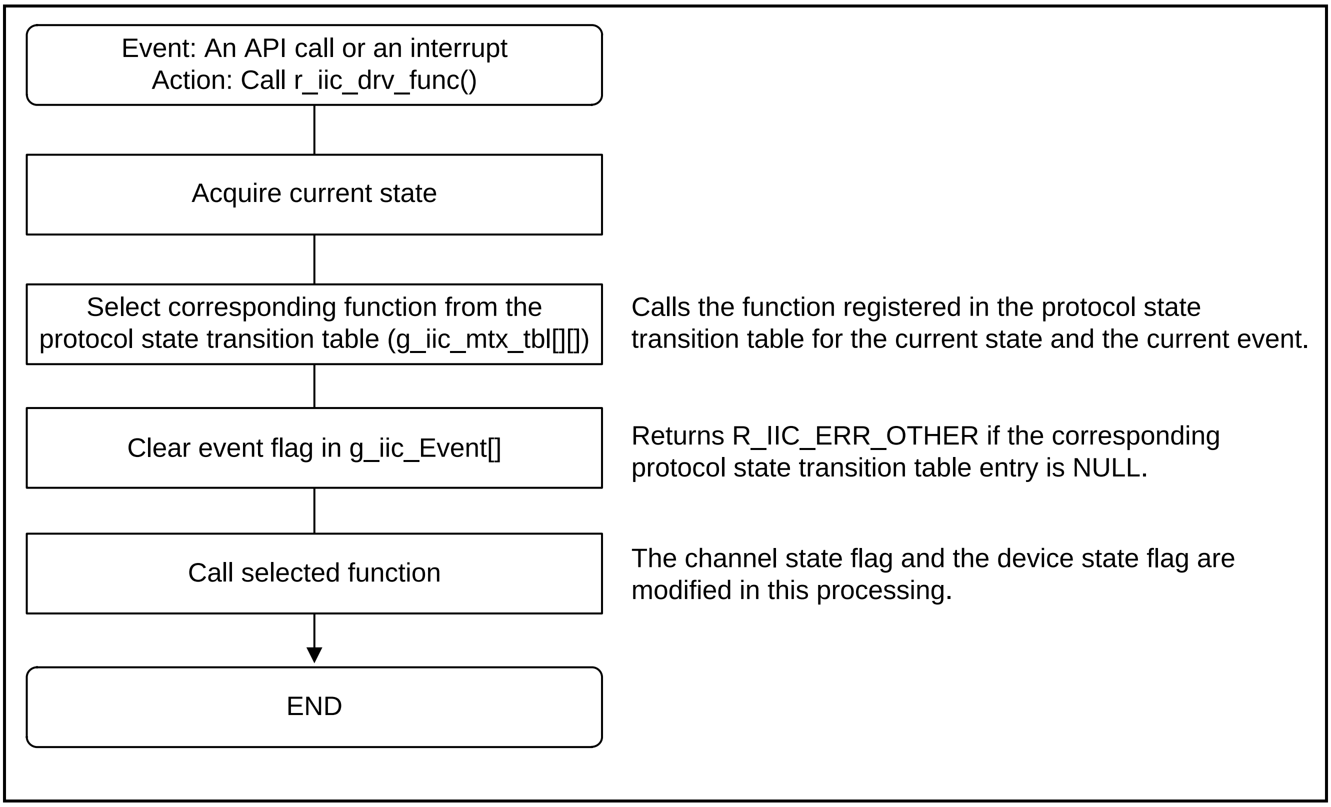


Figure 6.17 Communication Advance Processing Calling Mechanism

6.7 Interrupt Generation Timing

This section describes the interrupt timing in this driver.

Legend:

ST: Start condition

AD6-AD0: Slave address

/W: Transfer direction bit “0” (Write)

R: Transfer direction bit “1” (Read)

/ACK: Acknowledge “0”

NACK: Acknowledge “1”

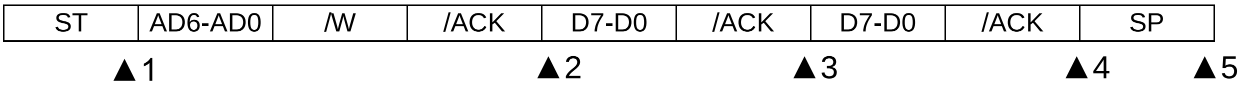
D7-D0: Data

RST: Restart condition

SP: Stop condition

6.7.1 Master Transmission

(1) Pattern 1



▲1: ICEEI (START) interrupt — start condition detected

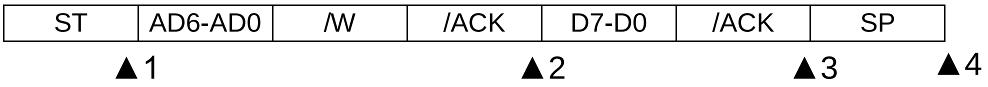
▲2: ICTEI interrupt — address transmission complete (transfer direction bit: write)*1

▲3: ICTEI interrupt — data transmission complete (1st data unit)*1

▲4: ICTEI interrupt — data transmission complete (2nd data unit)*1

▲5: ICEEI (STOP) interrupt — stop condition detected

(2) Pattern 2



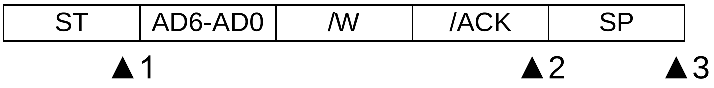
▲1: ICEEI (START) interrupt — start condition detected

▲2: ICTEI interrupt — address transmission complete (transfer direction bit: write)*1

▲3: ICTEI interrupt — data transmission complete (2nd data unit)*1

▲4: ICEEI (STOP) interrupt — stop condition detected

(3) Pattern 3

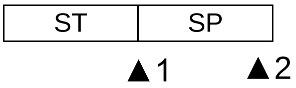


▲1: ICEEI (START) interrupt — start condition detected

▲2: ICTEI interrupt — address transmission complete (transfer direction bit: write)*[1]

▲3: ICEEI (STOP) interrupt — stop condition detected

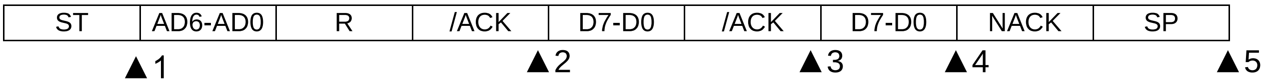
(4) Pattern 4



▲1: ICEEI (START) interrupt — start condition detected

▲2: ICEEI (STOP) interrupt — stop condition detected

6.7.2 Master Reception



▲1: ICEEI (START) interrupt — start condition detected

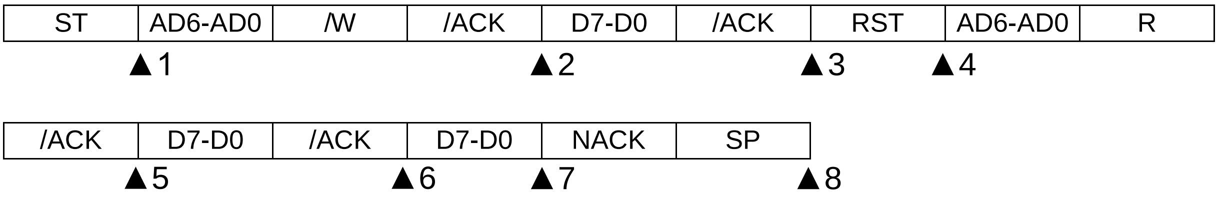
▲2: ICRXI interrupt — address transmission complete (transfer direction bit: Read)*[1]

▲3: ICRXI interrupt — Final data unit – 1 reception complete (2nd data unit)*[1]

▲4: ICRXI interrupt — Final data unit reception complete (2nd data unit)*[2]

▲5: ICEEI (STOP) interrupt — stop condition detected

6.7.3 Master Composite



▲1: ICEEI (START) interrupt — start condition detected

▲2: ICTEI interrupt — address transmission complete (transfer direction bit: write)*^{1}

▲3: ICTEI interrupt — data transmission complete (1st data unit)*^{1}

▲4: ICEEI (START) interrupt — restart condition detected

▲5: ICRXI interrupt — address transmission complete (transfer direction bit: Read)*^{1}

▲6: ICRXI interrupt — Final data unit – 1 reception complete (2nd data unit)*^{1}

▲7: ICRXI interrupt — Final data unit reception complete (2nd data unit)*^{2}

▲8: ICEEI (STOP) interrupt — stop condition detected

Notes: 1. Generated at the rise of the ninth clock pulse.
2. Generated at the rise of the eight clock pulse.

6.8 Callback Function

This function is called either if communication completes successfully or if it terminated with an error. To use this functionality, specify a function name for the CallBackFunc member of the I^{2}C communication information structure. See section 6.13.1, I^{2}C Communication Information Structure for details on this structure.

6.9 Relationship of Data Buffers and Transmit/Receive Data

The sample code is a block device driver, and transmit/receive data pointers are set as arguments. The relationship of the data alignment of the data buffers in RAM and the transmit/receive order is described below. Regardless of the endian mode or serial communication function used, data is transmitted in the transmit data buffer alignment order, and data is written to the receive data buffer in the order received.

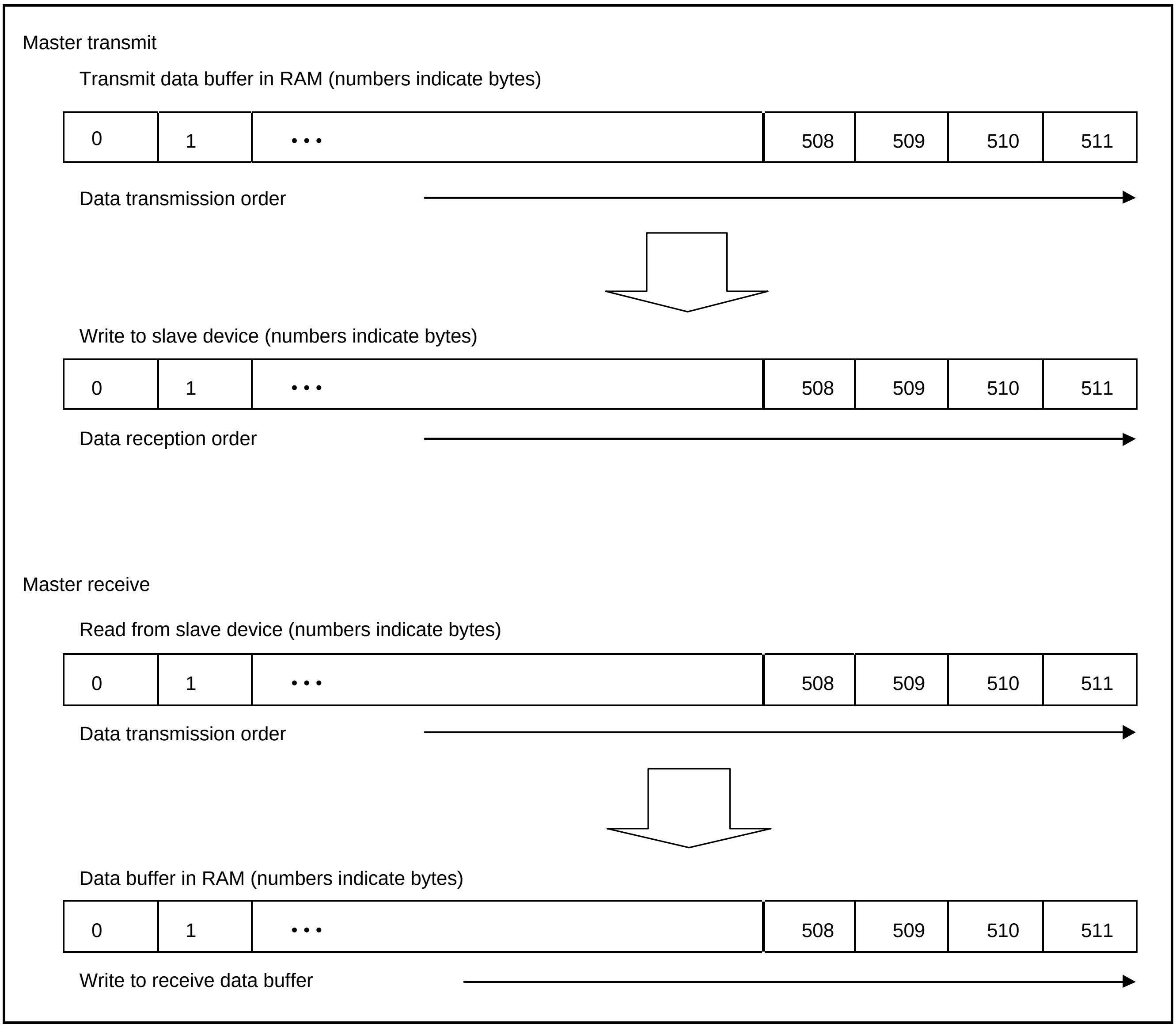


Figure 6.18 Storage of Transfer Data

6.10 Required Memory Sizes

The following lists the required memory sizes. The memory sizes listed below apply when one channel is used. The required memory sizes differ according to the number of channels used.

(1) RX63N

Table 6.7 Required Memory Sizes

Memory Used	Size	Remarks
ROM	4,648 bytes (little endian)	r_iic_drv_api.c r_iic_drv_int.c r_iic_drv_sfr.c r_iic_drv_sub.c
RAM	30 bytes (little endian)	r_iic_drv_api.c r_iic_drv_int.c r_iic_drv_sfr.c r_iic_drv_sub.c
Maximum usable user stack	84 bytes	
Maximum usable interrupt stack	4 bytes	

Note: The required memory sizes differ according to the C compiler version and the compile options.

(2) RX210

Table 6.8 Required Memory Sizes

Memory Used	Size	Remarks
ROM	4,654 bytes (little endian)	r_iic_drv_api.c r_iic_drv_int.c r_iic_drv_sfr.c r_iic_drv_sub.c
RAM	30 bytes (little endian)	r_iic_drv_api.c r_iic_drv_int.c r_iic_drv_sfr.c r_iic_drv_sub.c
Maximum usable user stack	84 bytes	
Maximum usable interrupt stack	4 bytes	

Note: The required memory sizes differ according to the C compiler version and the compile options.

6.11 File Structure

Table 6.11 lists the files used by the sample code. Note that files that are generated automatically by the integrated development environment are not listed.

Table 6.9 File Structure

\an_r01an1254ej0103_rx_iic				<DIR>	Sample code folder
	r01an1254ej0103_rx.pdf				Application note
	\source			<DIR>	Folder containing the program
		\r_iic_drv_rx		<DIR>	I^{2}C single master control software folder
			r_iic_drv_api.c		API source file
			r_iic_drv_api.h		API header file
			r_iic_drv_int.c		Interrupt handler source file
			r_iic_drv_int.h		Interrupt handler header file
			r_iic_drv_sfr.h.rx21a		Common register definitions header file (for the RX21A)
			r_iic_drv_sfr.h.rx62n		Common register definitions header file (for the RX62N)
			r_iic_drv_sfr.h.rx63n		Common register definitions header file (for the RX63N)
			r_iic_drv_sfr.h.rx63t		Common register definitions header file (for the RX63T)
			r_iic_drv_sfr.h.rx210		Common register definitions header file (for the RX210)
			r_iic_drv_sfr_rx21a.c		Common register definitions source file (for the RX21A)
			r_iic_drv_sfr_rx62n.c		Common register definitions source file (for the RX62N)
			r_iic_drv_sfr_rx63n.c		Common register definitions source file (for the RX63N)
			r_iic_drv_sfr_rx63t.c		Common register definitions source file (for the RX63T)
			r_iic_drv_sfr_rx210.c		Common register definitions source file (for the RX210)
			r_iic_drv_sub.c		Internal function source file
			r_iic_drv_sub.h		Internal function header file
		\sample		<DIR>	Folder containing the program for verifying EEPROM operation
			sample_background.c		Sample program for verifying EEPROM operation when calling the advance function from the RIIC interrupt handler
			sample_foreground.c		Sample program for verifying EEPROM operation when calling the advance function from the main processing routine

Note: A file with a filename of the form r_iic_drv_sfr.hXXX has been created for each microcontroller. One of these must be renamed to r_iic_drv_sfr.h and used. If there is no such file for the microcontroller used, the user must refer to these files and create an appropriate r_iic_drv_sfr.h file.

6.12 Constants

6.12.1 Return Values

The return value, channel state flag, and device state flag management values used in the sample code are listed below.

Table 6.10 Return Values, Channel State Flag, and Device State Flag Management Values (defined in r_iic_drv_api.h)

Constant Name	Setting Value	Description
R_IIC_NO_INIT	(error_t)(0)	Uninitialized state
R_IIC_IDLE	(error_t)(1)	Idle state: ready for communication
R_IIC_FINISH	(error_t)(2)	Idle state: previous communication complete, ready for communication
R_IIC_NACK	(error_t)(3)	Idle state: previous communication NACK complete, ready for communication
R_IIC_COMMUNICATION	(error_t)(4)	Communication in progress
R_IIC_LOCK_FUNC	(error_t)(5)	API processing in progress This state occurs in the following cases: • When another API function is called during API processing
R_IIC_BUS_BUSY	(error_t)(6)	Bus busy This state occurs in the following cases: • When, during communication, either the initialization function or a start function has been called • When another device is communicating over the same channel and either a start function or the advance function has been called
R_IIC_ERR_PARAM	(error_t)(-1)	Parameter error
R_IIC_ERR_AL	(error_t)(-2)	Arbitration lost error
R_IIC_ERR_NON_REPLY	(error_t)(-3)	No response error
R_IIC_ERR_SDA_LOW_HOLD	(error_t)(-4)	SDA held low error when SDL pseudo clock generate function called
R_IIC_ERR_OTHER	(error_t)(-5)	Other error

6.12.2 Definitions

The definitions of the constants used in the sample code are listed below, broken down by file.

Table 6.11 Constants Defined in r_iic_drv_api.h

Constant Name	Setting Value	Description
MAX_IIC_CH_NUM	(uint8_t)(1)	One plus the maximum number of channels that can be used at the same time. Sets to 1 in this sample code.
REPLY_CNT	(uinit32_t)(10000)	Advanced function counter*1
STOP_COND_WAIT	(uint16_t)(1000)	Stop condition generation wait counter*1
BUSCHK_CNT	(uint16_t)(1000)	Bus busy check counter*1
SDACHK_CNT	(uint16_t)(1000)	SDA level check counter*1
GEN_SCLCLK_WAIT	(uint16_t)(1000)	Pseudo clock generation wait counter*1
W_CODE	(uint8_t)(0x00)	Setting value when slave address transfer direction is “write”
R_CODE	(uint8_t)(0x01)	Setting value when slave address transfer direction is “Read”
R_IIC_HI	(uint8_t)(0x01)	Port “H”
R_IIC_LOW	(uint8_t)(0x00)	Port “L”
R_IIC_OUT	(uint8_t)(0x01)	Port Output
R_IIC_IN	(uint8_t)(0x00)	Port Input
R_IIC_FALSE	(uint8_t)(0x00)	Flag "OFF"
R_IIC_TRUE	(uint8_t)(0x01)	Flag "ON"

Note: 1. Counter value settings

These are counters for software loops. This means that the loop time will depend on the system clock actually used. These values must be set according to the system clock used.

Table 6.12 Constants Defined in r_iic_drv_sfr.h.rxXXX
(XXX represents the microcontroller model number.)

Constant Name	Setting Value	Description
R_IIC_CHx_LCLK	(uint8_t)(0xED)	I^{2}C bus bit rate low-level register (ICBRL) setting for channel x (x = channel number)*2
R_IIC_CHx_HCLK	(uint8_t)(0xE6)	I^{2}C bus bit rate high-level register (ICBRL) setting for channel x (x = channel number)*2
R_IIC_CHx_ICMR1_INIT	(uint8_t)(0x28)	I^{2}C bus mode register 1 (ICMR1) setting for channel x (x = channel number)*2
R_IIC_IPR_CHx_EEI_INIT	(uint8_t)(0x02)	ICEEIx interrupt priority for channel x (x = channel number)
R_IIC_IPR_CHx_RXI_INIT	(uint8_t)(0x02)	ICRXIx interrupt priority for channel x (x = channel number)
R_IIC_IPR_CHx_TXI_INIT	(uint8_t)(0x02)	ICTXIx interrupt priority for channel x (x = channel number)
R_IIC_IPR_CHx_TEI_INIT	(uint8_t)(0x02)	ICTEIx interrupt priority for channel x (x = channel number)

Note: 2. Transfer rate setting

The transfer clock is determined by the following settings. These settings must be made for each channel used. Define each setting as needed. For details of the setting procedure, see the RX Family User’s Manual: Hardware.

- Internal reference clock select bits (CKS[2:0]) in I^{2}C bus mode register 1 (ICMR1)
- I^{2}C bus bit rate low-level register (ICBRL)
- I^{2}C bus bit rate high-level register (ICBRH)

The maximum setting is 400 kHz. However, if standard mode devices and fast mode devices are used together, the standard mode maximum rate of 100 kHz must be used as the setting. Note that it may be necessary to modify the setting value since the rise time (tR) and fall time (tF) of the SDA and SCL signals differ according to the pull-up resistance and the wiring capacitance.

6.13 Structures and Unions

6.13.1 I^{2}C Communication Information Structure

The figure below shows the I^{2}C communication information structure used in the sample code. An instance of this structure must be set up for each slave device used.

```
typedef struct
{
  uint8_t             *pSlvAdr;          /* Pointer for Slave address buffer            */
  uint8_t             *pData1st;         /* Pointer for 1st Data buffer                 */
  uint8_t             *pData2nd;         /* Pointer for 2nd Data buffer                 */
  error_t             *pDevStatus;       /* Device status flag                          */
  uint32_t            Cnt1st;            /* 1st Data counter                            */
  uint32_t            Cnt2nd;            /* 2nd Data counter                            */
  r_iic_callback      CallBackFunc;      /* Callback function                           */
  uint8_t             ChNo;              /* Channel No.                                 */
  uint8_t             rsv1;
  uint8_t             rsv2;
  uint8_t             rsv3;
} r_iic_drv_info_t;
```

Figure 6.19 I^{2}C Communication Information Structure

(1) Structure Members

The table below lists the structure members. See tables 6.16 and 6.17 for details on setting the r_iic_drv_info_t members.

Table 6.13 Structure r_iic_drv_info_t Members

Structure member	Description
*pSlvAdr	Slave address storage buffer pointer Allocate one byte for this data.
*pData1st	1st data storage buffer pointer
*pData2nd	2nd data storage buffer pointer
*pDevStatus	Device state flag pointer Device states can be checked during communication, even when multiple devices are connected to the same channel. Allocate one byte for this data. See section 8.6 for a usage example.
Cnt1st	1st data counter (byte count)
Cnt2nd	2nd data counter (byte count)
CallBackFunc	Callback function
ChNo	Channel number of the used device Set this to the channel number of the bus used.
rsv1 rsv2 rsv3	Alignment adjustment members

(2) Settings

Table 6.16 lists the allowable range of user settings for the structure r_iic_drv_info_t members for master transmission and table 6.17 lists those for master reception and master composite.

Table 6.14 User Setting Ranges for r_iic_drv_info_t Members: Master Transmission

Structure Member	Allowable User Setting Range			
	Master Transmission Pattern 1	Master Transmission Pattern 2	Master Transmission Pattern 3	Master Transmission Pattern 4
*pSlvAdr	Slave address storage source address	Slave address storage source address	Slave address storage source address	NULL
*pData1st	1st data storage source address	NULL	NULL	NULL
*pData2nd	2nd data (transmit data) storage source address	Second data (transmit data) storage source address	NULL	NULL
*pDevStatus	Device state storage source address	Device state storage source address	Device state storage source address	Device state storage source address
Cnt1st	0000 0001h*[1] to FFFF FFFFh	(Invalid setting)	(Invalid setting)	(Invalid setting)
Cnt2nd	0000 0001h*[1] to FFFF FFFFh	0000 0001h*[1] to FFFF FFFFh	(Invalid setting)	(Invalid setting)
CallBackFunc	If used: specify the name of the function. If not, set to NULL.	If used: specify the name of the function. If not, set to NULL.	If used: specify the name of the function. If not, set to NULL.	If used: specify the name of the function. If not, set to NULL.
ChNo	00h to FFh	00h to FFh	00h to FFh	00h to FFh
rsv1, rsv2, rsv3	(Invalid setting)	(Invalid setting)	(Invalid setting)	(Invalid setting)

Table 6.15 User Setting Ranges for r_iic_drv_info_t Members: Master Reception and Master Composite

Structure Member	Allowable User Setting Range	
	Master Reception	Master Composite
*pSlvAdr	Slave address storage source address	Slave address storage source address
*pData1st	(Invalid setting)	1st data storage source address
*pData2nd	2nd data (receive data) storage destination address	2nd data (receive data) storage destination address
*pDevStatus	Device state storage source address	Device state storage source address
Cnt1st	(Invalid setting)	0000 0001h*[1] to FFFF FFFFh
Cnt2nd	0000 0001h*[1] to FFFF FFFFh	0000 0001h*[1] to FFFF FFFFh
CallBackFunc	If used: specify the name of the function. If not, set to NULL.	If used: specify the name of the function. If not, set to NULL.
ChNo	00h to FFh	00h to FFh
rsv1, rsv2, rsv3	(Invalid setting)	(Invalid setting)

Note: 1. The value 0 is illegal in both table 6.16 and table 6.17.

(3) Callback Function

This function is called either if communication completes successfully or if it terminated with an error. To use this functionality, specify a function name for the CallBackFunc member.

(4) Notes On Settings

During master transmission, the data stored in the members of this structure is referenced to determine what operation to perform. This sample code may fail to operate correctly if any values other than those listed in table 6.16 are used.

6.13.2 Internal Information Management Structure

The figure below shows the internal information management structure used by the sample code. Since this structure is controlled by the sample code, there is no need for it to be set by the user.

```
typedef struct
{
  r_iic_drv_internal_mode_t        Mode;          /* Mode of Control Protocol          */
  r_iic_drv_internal_status_t      N_status;      /* Internal Status of NOW            */
  r_iic_drv_internal_status_t      B_status;      /* Internal Status of BEFORE         */
} r_iic_drv_internal_info_t;
```

Figure 6.20 Internal information management structure

(1) Structure Members

The table lists the structure members.

Table 6.16 Structure r_iic_drv_internal_info_t Members

Structure Member	Description
Mode	I²C protocol mode See table 6.19 for the definition of the data stored.
N_status	The protocol control current state. Values defined in table 6.2 are stored in this member.
B_status	The protocol control previous state. Values defined in table 6.2 are stored in this member.

6.14 Enumerated Types

The enumerated type definitions used in the sample code are listed below.

Table 6.17 I^{2}C Protocol Operating Modes (enum r_iic_drv_internal_mode_t)

	Description
R_IIC_MODE_NONE	No communication state This mode is transitioned to from the uninitialized state or from the idle state.
R_IIC_MODE_WRITE	Master transmission in progress This mode is transitioned to by starting communication with the master transmission start function R_IIC_Drv_MasterTx().
R_IIC_MODE_READ	Master reception in progress This mode is transitioned to by starting communication with the master reception start function R_IIC_Drv_MasterRx().
R_IIC_MODE_COMBINED	Master composite operation in progress This mode is transitioned to by starting communication with the master composite start function R_IIC_Drv_MasterTRx().

6.15 Variables

Table 6.20 lists the global variable.

Table 6.18 Global Variable

Type	Valuable	Description	Function Used
uint8_t	g_iic_ChStatus[MAX_IIC_CH_NUM]	Channel state flag The communication state defined in table 6.12 can be checked. Set this variable to R_IIC_NO_INIT at initialization.	R_IIC_Drv_Init R_IIC_Drv_MasterTx R_IIC_Drv_MasterRx R_IIC_Drv_MasterTRx R_IIC_Drv_Advance R_IIC_Drv_GenClk R_IIC_Drv_Reset
r_iic_drv_internal_event_t	g_iic_Event[MAX_IIC_CH_NUM]	Event flag This flag is set when an interrupt occurs and is cleared by the advance function. The value after clearing is R_IIC_EV_INIT. See table 6.3 for the setting values, and see figure 6.9 for the relationship between setting and clearing.	R_IIC_Drv_Init R_IIC_Drv_MasterTx R_IIC_Drv_MasterRx R_IIC_Drv_MasterTRx R_IIC_Drv_Advance
r_iic_drv_internal_info_t	g_iic_InternalInfo[MAX_IIC_CH_NUM]	Internal information management This variable is managed by the sample code and must not be set by the user.	R_IIC_Drv_Init R_IIC_Drv_MasterTx R_IIC_Drv_MasterRx R_IIC_Drv_MasterTRx R_IIC_Drv_Advance
uint32_t	g_iic_ReplyCnt[MAX_IIC_CH_NUM]	Advance function counter This is the upper limit on the number of calls the advance function. It is decremented by the advance function called by the user. It is initialized when an event occurs. If it reaches 0, the channel state flag and the device state flag are set to R_IIC_ERR_NON_REPLY. The counter value can be modified with the REPLY_CNT macro definition. This macro should be set appropriately for the actual user system.	R_IIC_Drv_Init R_IIC_Drv_MasterTx R_IIC_Drv_MasterRx R_IIC_Drv_MasterTRx R_IIC_Drv_Advance
bool	g_iic_Api[MAX_IIC_CH_NUM]	API flag This flag is used to prevent multiple calls this sample code's API. It is set when API processing starts and cleared after that processing completes.	R_IIC_Drv_Init R_IIC_Drv_MasterTx R_IIC_Drv_MasterRx R_IIC_Drv_MasterTRx R_IIC_Drv_Advance R_IIC_Drv_GenClk R_IIC_Drv_Reset

6.16 Functions

Table 6.21 lists the Functions.

Table 6.19 Functions

Function	Description
R_IIC_Drv_Init()	I^{2}C driver initialization function
R_IIC_Drv_MasterTx()	Master transmission start function
R_IIC_Drv_MasterRx()	Master reception start function
R_IIC_Drv_MasterTRx()	Master composite start function
R_IIC_Drv_Advance()	Advance function
R_IIC_Drv_GenClk()	SCL pseudo clock generation function
R_IIC_Drv_Reset()	I^{2}C driver reset function

6.17 State Transition Diagram

Figure 6.21 is a diagram showing state transitions for each channel.

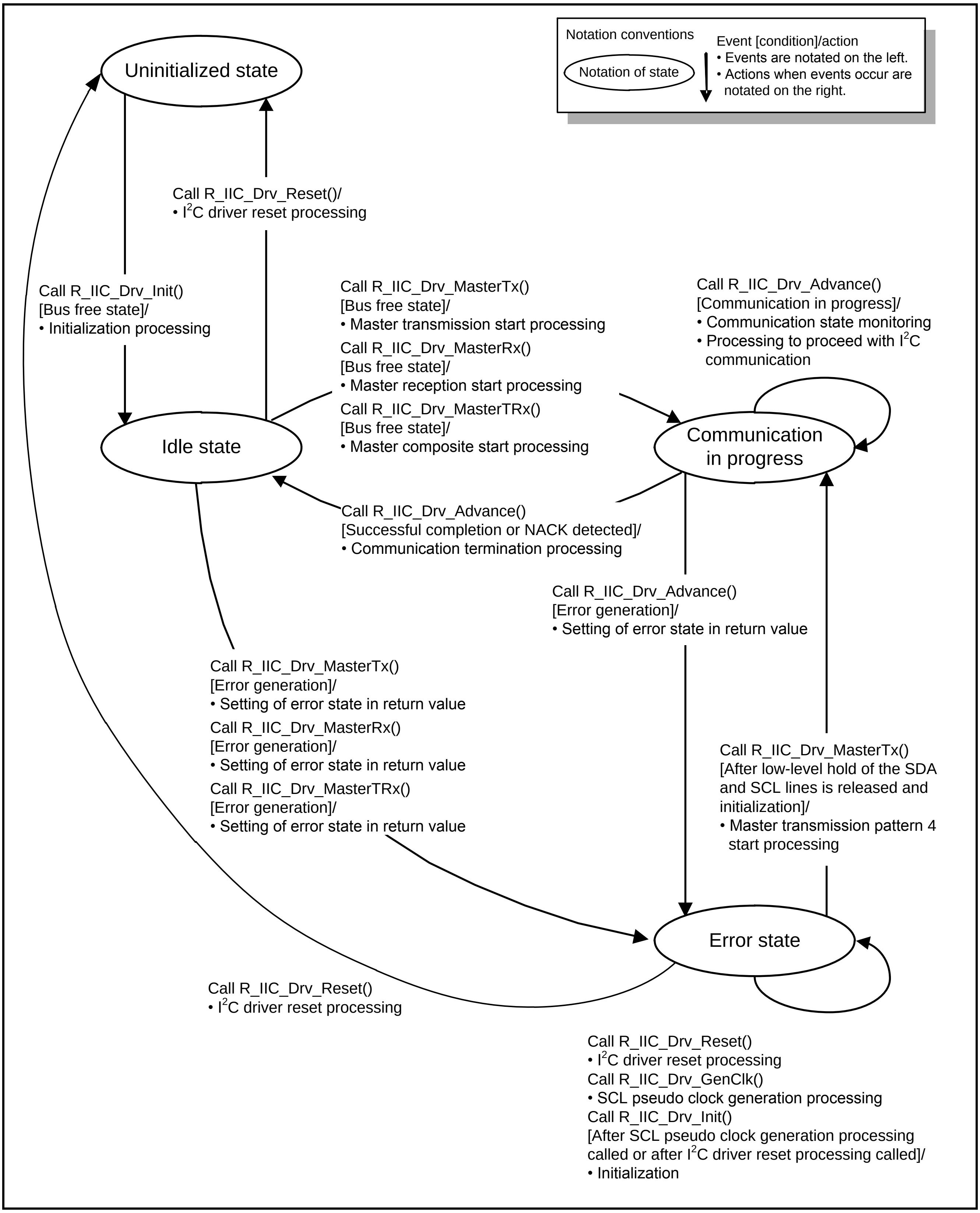


Figure 6.21 State Transition Diagram

6.17.1 Error State Definitions

In this sample code, occurrences of the following phenomena are defined to be error states. Error occurrences can be verified from the return values after return from the API functions. See the return values from each of the API functions in section 6.18, Function Specifications, for methods for responding when an error occurs.

(1) Parameter Error

Return value: R_IIC_ERR_PARAM

If the arguments were not set appropriately when an API function was called.

(2) Arbitration Lost

Return value: R_IIC_ERR_AL

If arbitration was lost. See the RX Family User’s Manual: Hardware for the conditions where this occurs.

(3) No Response Error

Return value: R_IIC_NON_REPLY

The following cases result in a no response error.

- If the number of advance function calls exceeds the limit
- When a start function was called, if the bus was monitored for a fixed time but was not released
- If the start condition generation processing was performed but it was not detected after a fixed time had passed
- If the stop condition generation processing was performed when the advance function was called but it was not detected after a fixed time had passed

Note that the start condition generation wait time is measured with a software loop. The counter value can be set by the user. Set this value according to the system clock used. See table 6.13 for the definition of this counter.

(4) SDA Held Low (Recovery not possible)

Return value: R_IIC_ERR_SDA_LOW_HOLD

If an SCL pseudo clock was generated but SDA remained held at the low level.

(5) Other Errors

Return value: R_IIC_ERR_OTHER

If an error other than (1) to (4) above occurred.

6.17.2 Flag States at State Transitions

Table 6.22 lists the states of the flags when a state transition occurs.

Table 6.20 Flag States at State Transitions

State	Channel State Flag	Device State Flag (Communicating Device)	I^{2}C Protocol Operating Mode	Current State of The Protocol Control
	g_iic_ChStatus[]	I^{2}C Communication Information Structure *pDevStatus	Internal Communication Information Structure Mode	Internal Communication Information Structure N_status
Uninitialized state	R_IIC_NO_INIT	R_IIC_NO_INIT	R_IIC_MODE_NONE	R_IIC_STS_NO_INIT
Idle state	R_IIC_IDLE R_IIC_FINISH R_IIC_NACK	R_IIC_IDLE R_IIC_FINISH R_IIC_NACK	R_IIC_MODE_NONE	R_IIC_STS_IDLE
Communication in progress (master transmission)	R_IIC_COMMUNICATION	R_IIC_COMMUNICATION	R_IIC_MODE_WRITE	R_IIC_STS_ST_COND_WAIT
				R_IIC_STS_SEND_SLVADR_W_WAIT
				R_IIC_STS_SEND_SLVADR_R_WAIT
				R_IIC_STS_SEND_DATA_WAIT
				R_IIC_STS_RECEIVE_DATA_WAIT
				R_IIC_STS_SP_COND_WAIT
Communication in progress (master reception)	R_IIC_COMMUNICATION	R_IIC_COMMUNICATION	R_IIC_MODE_READ	R_IIC_STS_ST_COND_WAIT
				R_IIC_STS_SEND_SLVADR_W_WAIT
				R_IIC_STS_SEND_SLVADR_R_WAIT
				R_IIC_STS_SEND_DATA_WAIT
				R_IIC_STS_RECEIVE_DATA_WAIT
				R_IIC_STS_SP_COND_WAIT
Communication in progress (master composite)	R_IIC_COMMUNICATION	R_IIC_COMMUNICATION	R_IIC_MODE_COMBINED	R_IIC_STS_ST_COND_WAIT
				R_IIC_STS_SEND_SLVADR_W_WAIT
				R_IIC_STS_SEND_SLVADR_R_WAIT
				R_IIC_STS_SEND_DATA_WAIT
				R_IIC_STS_RECEIVE_DATA_WAIT
				R_IIC_STS_SP_COND_WAIT
Error state	R_IIC_ERR_PARAM	R_IIC_ERR_PARAM	—	—
	R_IIC_ERR_AL	R_IIC_ERR_AL	—	—
	R_IIC_ERR_NON_REPLY	R_IIC_ERR_NON_REPLY	—	—
	R_IIC_ERR_SCL_GENCLK	R_IIC_ERR_SCL_GENCLK	—	—
	R_IIC_ERR_OTHER	R_IIC_ERR_OTHER	—	—

6.18 Function Specifications

6.18.1 Common Processing for These Functions

This sample code has an API that can be operated once. If this sample code's API is called during execution of this API processing, the processing is not performed and the function terminates. The value R_IIC_LOCK_FUNC is returned in this case.

An API flag is provided to prevent simultaneous calls the API. This flag is set while API processing is being performed. This mechanism operates as follows: at the start of API processing the flag is checked and the processing is only performed if the flag is not set. Figure 6.22 presents an overview of this processing as flowcharts.

This processing is performed for the functions defined in section 6.16. The subsequent processing indicated as "user API processing" in figure 6.22 is described in section 6.18.2 and the following sections.

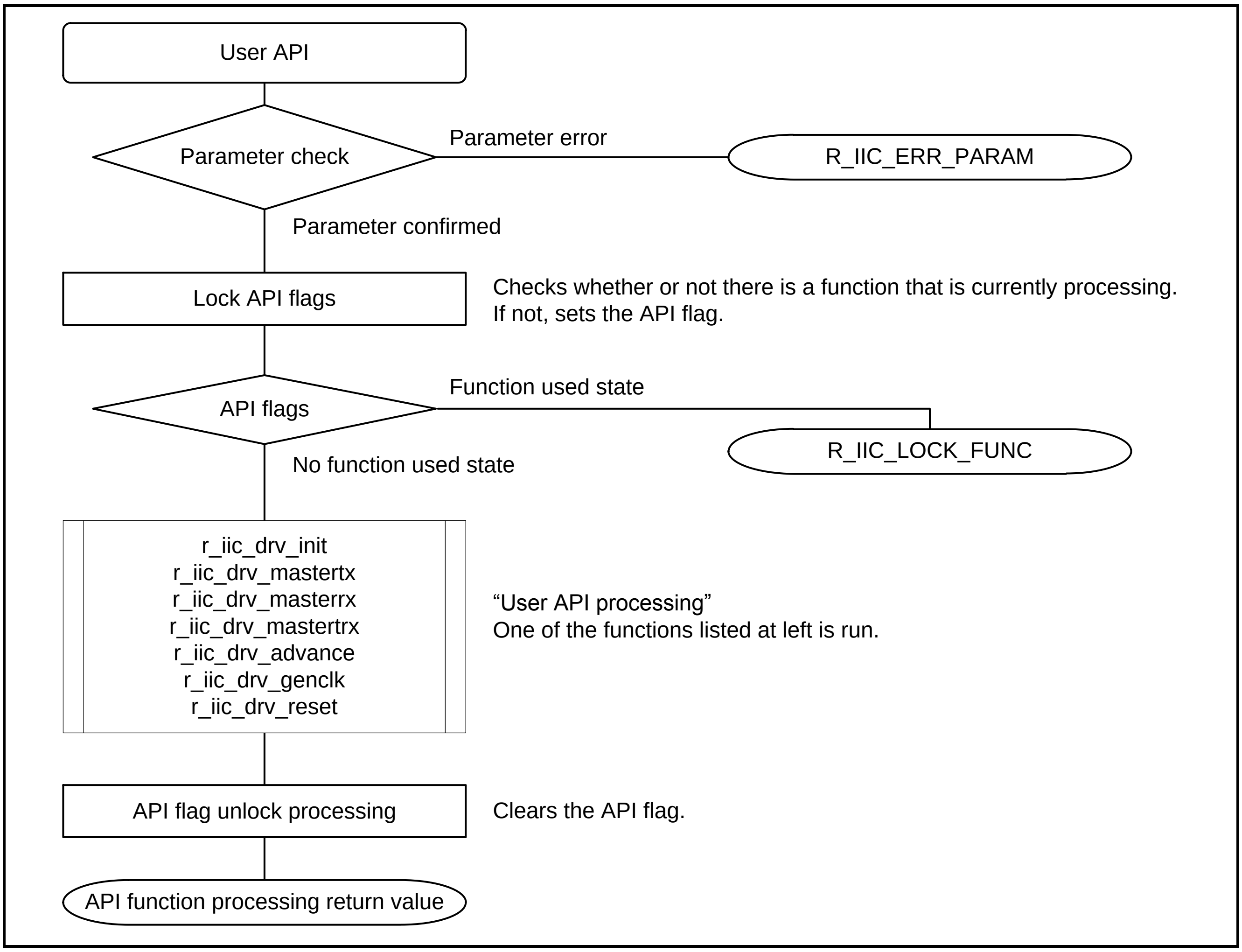


Figure 6.22 Simplified Flowchart of Processing to Prevent Multiple Overlapping Function Calls

6.18.2 I^{2}C Driver Initialization Function

R_IIC_Drv_Init	
Outline	I^{2}C driver initialization function
Header	r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h
Declaration	error_t R_IIC_Drv_Init(r_iic_drv_info_t *pRIic_Info)
Description	• Initializes the corresponding channel. • The following must be set up to use this function. The ChNo member of the r_iic_drv_info_t structure; The channel number used The channel state flag (g_iic_ChStatus[]); Sets R_IIC_NO_INIT*1 The device state flag (*(pRIic_Info.pDevStatus)); Sets R_IIC_NO_INIT*1
Arguments	r_iic_drv_info_t *pRIic_Info ; Pointer to I^{2}C communication information structure
Return Value	R_IIC_IDLE In the channel uninitialized state, this function performs the initialization and transitions to the idle state. The channel state flag and device state flag are set to R_IIC_IDLE. In the already initialized state, initialization is not performed and the device state flag is set to R_IIC_IDLE. → Communication is now possible by calling the start function. R_IIC_FINISH / R_IIC_NACK This is the result of executing the preprocessing. Since the start function can be called, initialization is not performed. The channel state flag and device state flag are not changed. → Communication is now possible by calling the start function. R_IIC_LOCK_FUNC The processing was not performed because another API operation was being performed. The channel state flag and device state flag are not changed. → Call the function after processing of the other API finishes. R_IIC_BUS_BUSY Communication is in progress. Initialization is not possible. The channel state flag and device state flag are not changed. → Call the advance function to terminate communication. R_IIC_ERR_PARAM A parameter error was detected. The arguments were not set up. The channel state flag and device state flag are not changed. → Set up the arguments as required by this function. R_IIC_ERR_AL Arbitration was lost. The channel state flag and device state flag are not changed. → See section 7.3, Recovery Processing Example, and perform that recovery processing. R_IIC_ERR_NON_REPLY A no replay error occurred. The channel state flag and device state flag are not changed. → See section 7.3, Recovery Processing Example, and perform that recovery processing. R_IIC_ERR_SDA_LOW_HOLD SDA is in the state where it has not recovered from the low-level hold state. The channel state flag and device state flag are not changed. → Check the system states, including whether a slave device is holding SDA low and whether a low-level signal has not been output from the master device.

R_IIC_ERR_OTHER

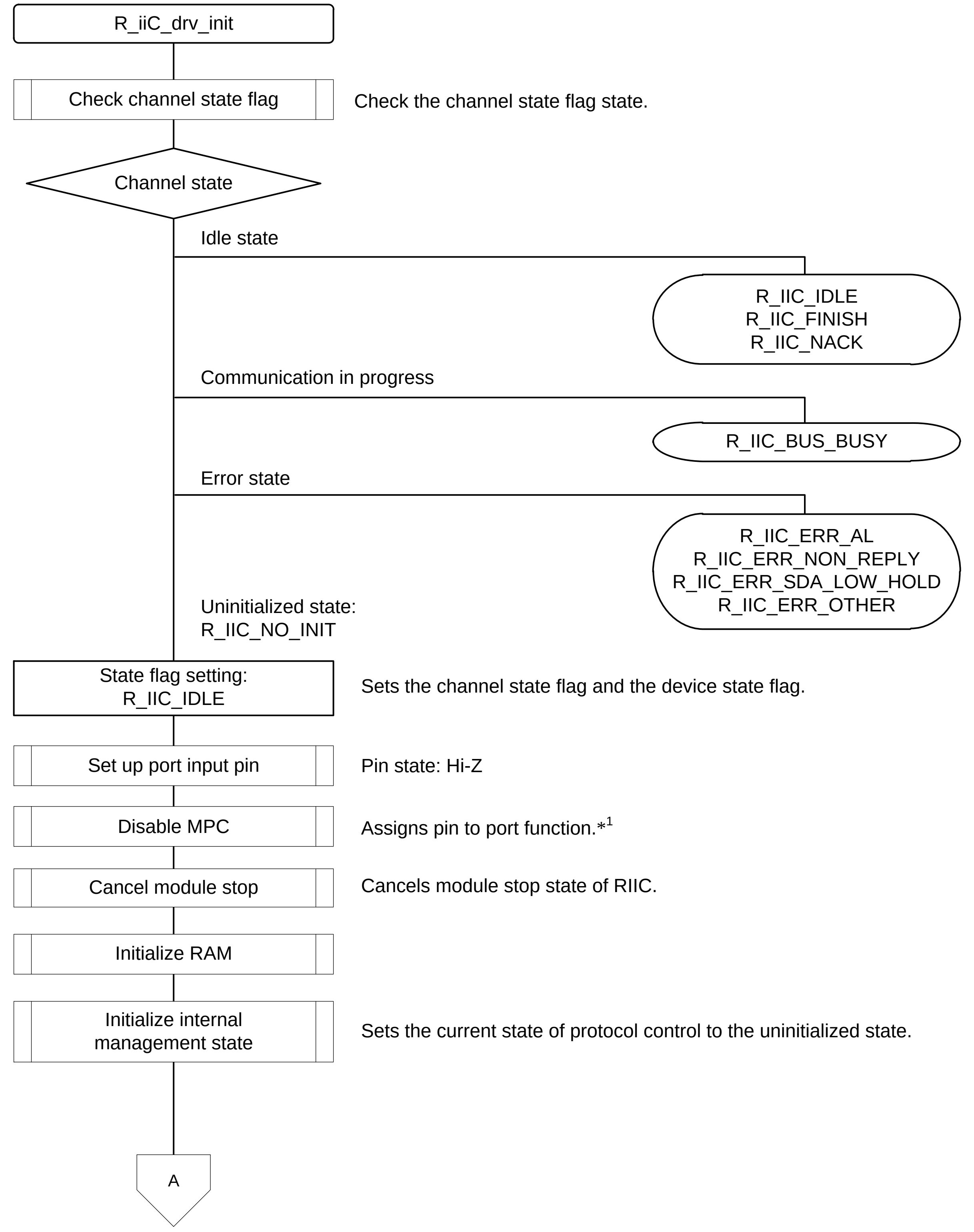
Some other error occurred. The channel state flag and device state flag are set to R_IIC_ERR_OTHER.

If an error had already occurred, no processing is performed. The channel state flag and device state flag are not changed.

→ Check the following items.

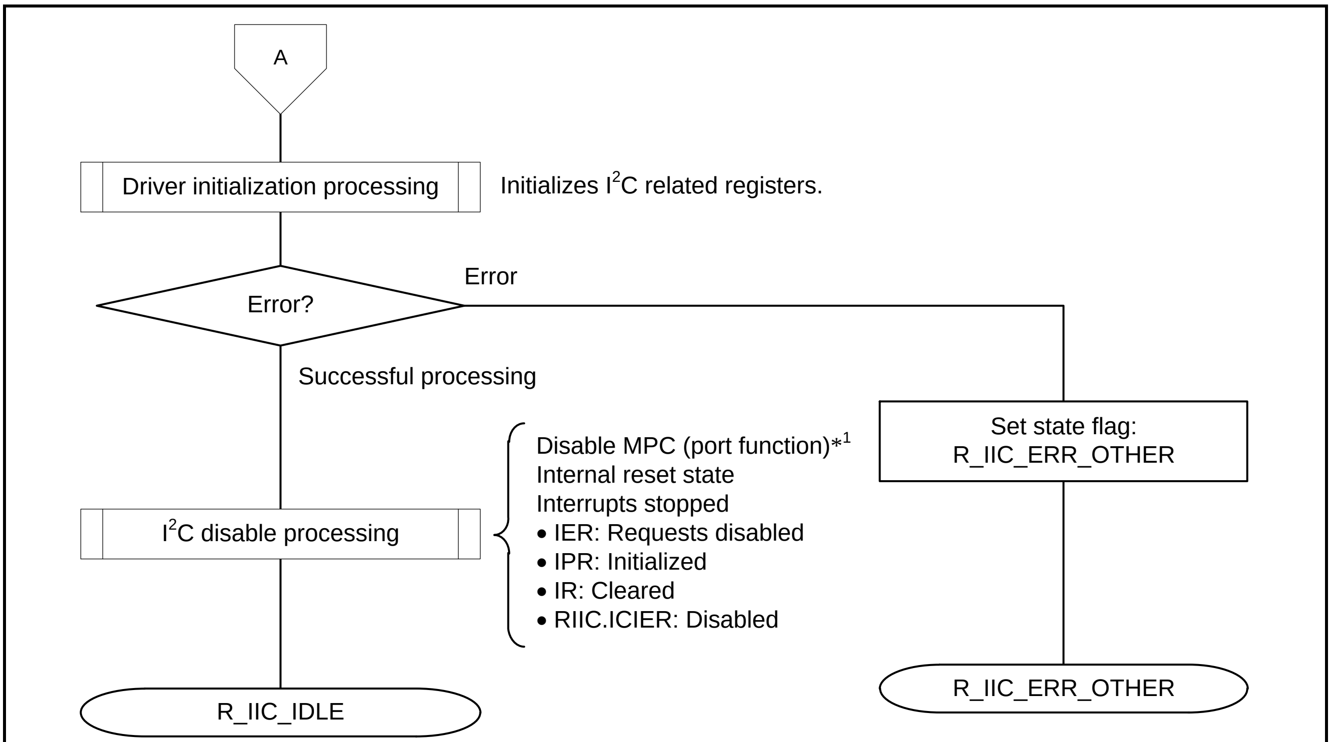
— Check that the I^{2}C communication information structure is set up correctly.

Remarks Note: 1. Before calling the initialization function, set R_IIC_NO_INIT. If the initialization function is called without setting this, the initialization processing may not be performed.



Note: 1. If an MPC is present, the multi-function pin controller setting is disabled.
If no MPC is present, the pin is disabled in the input buffer control register.

Figure 6.23 I^{2}C Driver Initialization Function Overview Flowchart (1/2)



Note: 1. If an MPC is present, the multi-function pin controller setting is disabled.
If no MPC is present, the pin is disabled in the input buffer control register.

Figure 6.24 I²C Driver Initialization Function Overview Flowchart (2/2)

6.18.3 Master Transmission Start Function

R_IIC_Drv_MasterTx	
Outline	Master transmission start function
Header	r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h
Declaration	error_t R_IIC_Drv_MasterTx (r_iic_drv_info_t *pRIic_Info)
Description	• Starts master transmission. • The r_iic_drv_info_t I^{2}C communication information structure must be set up to perform this operation. See table 6.16 for details on that setup.
Arguments	r_iic_drv_info_t *pRIic_Info ; Pointer to I^{2}C communication information structure
Return Value	R_IIC_COMMUNICATION Master transmission started. The channel state flag and device state flag are set to R_IIC_COMMUNICATION. → Call the advance function to terminate communication. R_IIC_NO_INIT Initialization was not performed.*1 The channel state flag and device state flag are not changed. → Call the initialization function and assure its processing has completed. R_IIC_LOCK_FUNC The processing was not performed because another API operation was being performed. The channel state flag and device state flag are not changed. → Call the function after processing of the other API finishes. R_IIC_BUS_BUSY Communication is in progress. It was not possible to start master transmission. The channel state flag and device state flag are not changed. → Call the advance function to terminate communication. R_IIC_ERR_PARAM A parameter error was detected. The arguments were not set up. The channel state flag and device state flag are not changed. → Set up the arguments as required by this function. R_IIC_ERR_AL Arbitration was lost. The channel state flag and device state flag are not changed. → See section 7.3, Recovery Processing Example, and perform that recovery processing. R_IIC_ERR_NON_REPLY Either the bus was not released or it was not possible to detect the start condition. The channel state flag and device state flag are set to R_IIC_ERR_NON_REPLY. → See section 7.3, Recovery Processing Example, and perform that recovery processing. R_IIC_ERR_SDA_LOW_HOLD SDA is in the state where it has not recovered from the low-level hold state. The channel state flag and device state flag are not changed. → Check the system states, including whether a slave device is holding SDA low and whether a low-level signal has not been output from the master device. R_IIC_ERR_OTHER Some other error occurred. The channel state flag and device state flag are set to R_IIC_ERR_OTHER. If an error had already occurred, no processing is performed. The channel state flag and device state flag are not changed. → Check the following items. — Check that the I^{2}C communication information structure is set up correctly.

Remarks

- At the point this function returns, I^{2}C communication has not completed. The advance function must be called to terminate I^{2}C communication.
- The communication state after calling the start function can be checked with the return value from the advance function.

Note: 1. Even if initialization was performed once, the driver may enter the uninitialized state if another device on the same channel subsequently calls the I^{2}C driver setting function. In such cases, call the I^{2}C driver initialization function once again before calling the start function.

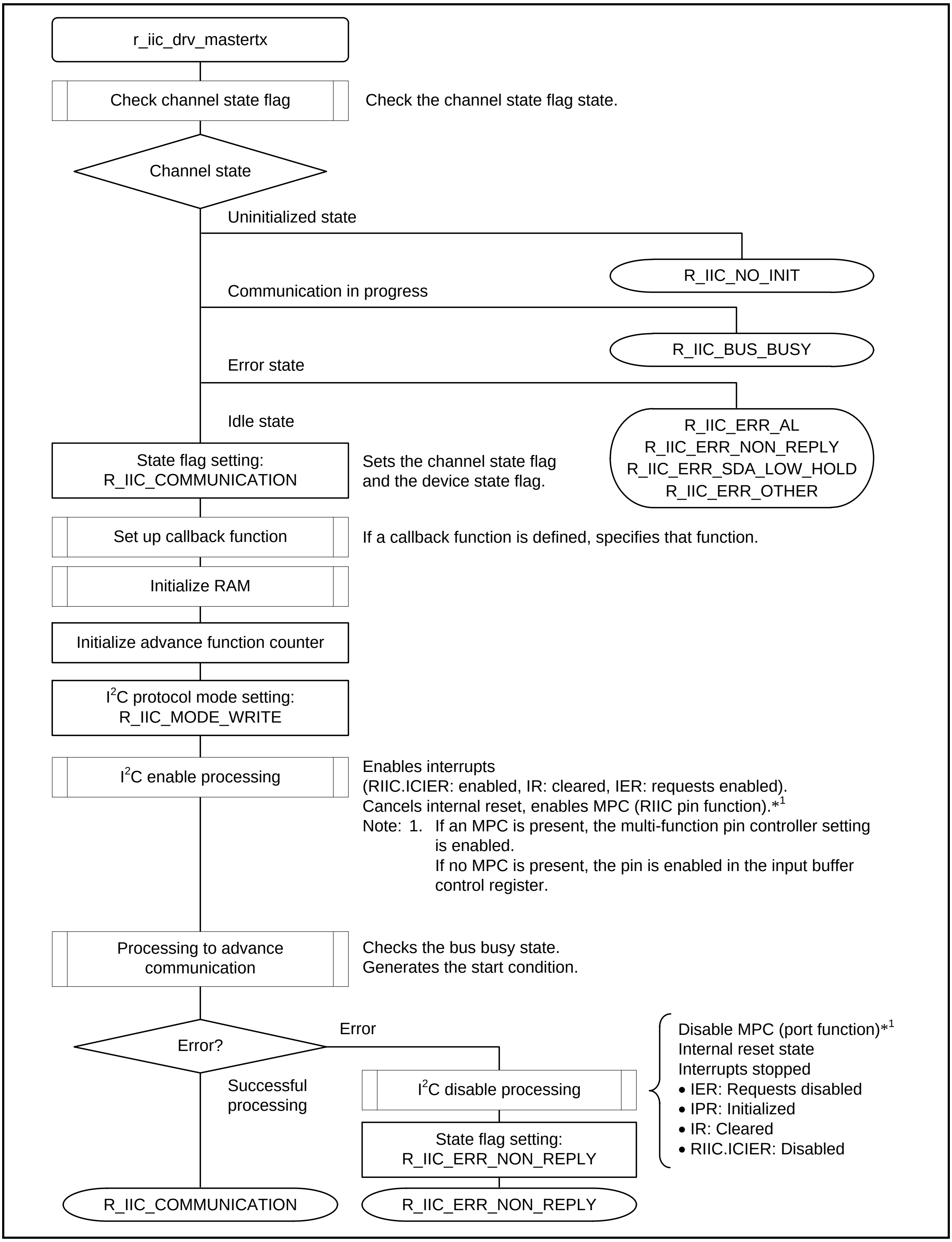


Figure 6.25 Master Transmission Start Function Overview Flowchart

6.18.4 Master Reception Start Function

R_IIC_Drv_MasterRx

Outline	Master reception start function
Header	r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h
Declaration	error_t R_IIC_Drv_MasterRx (r_iic_drv_info_t *pRIic_Info)
Description	• Starts master reception. • The r_iic_drv_info_t I^{2}C communication information structure must be set up to perform this operation. See table 6.17 for details on that setup.
Arguments	r_iic_drv_info_t *pRIic_Info ; Pointer to I^{2}C communication information structure
Return Value	R_IIC_COMMUNICATION Master reception started. The channel state flag and device state flag are set to R_IIC_COMMUNICATION. → Call the advance function to terminate communication. R_IIC_NO_INIT Initialization was not performed.*1 The channel state flag and device state flag are not changed. → Call the initialization function and assure its processing has completed. R_IIC_LOCK_FUNC The processing was not performed because another API operation was being performed. The channel state flag and device state flag are not changed. → Call the function after processing of the other API finishes. R_IIC_BUS_BUSY Communication is in progress. It was not possible to start master reception. The channel state flag and device state flag are not changed. → Call the advance function to terminate communication. R_IIC_ERR_PARAM A parameter error was detected. The arguments were not set up. The channel state flag and device state flag are not changed. → Set up the arguments as required by this function. R_IIC_ERR_AL Arbitration was lost. The channel state flag and device state flag are not changed. → See section 7.3, Recovery Processing Example, and perform that recovery processing. R_IIC_ERR_NON_REPLY Either the bus was not released or it was not possible to detect the start condition. The channel state flag and device state flag are set to R_IIC_ERR_NON_REPLY. → See section 7.3, Recovery Processing Example, and perform that recovery processing. R_IIC_ERR_SDA_LOW_HOLD SDA is in the state where it has not recovered from the low-level hold state. The channel state flag and device state flag are not changed. → Check the system states, including whether a slave device is holding SDA low and whether a low-level signal has not been output from the master device. R_IIC_ERR_OTHER Some other error occurred. The channel state flag and device state flag are set to R_IIC_ERR_OTHER. If an error had already occurred, no processing is performed. The channel state flag and device state flag are not changed. → Check the following items. — Check that the I^{2}C communication information structure is set up correctly.

Remarks

- At the point this function returns, I^{2}C communication has not completed. The advance function must be called to terminate I^{2}C communication.
- The communication state after calling the start function can be checked with the return value from the advance function.

Note: 1. Even if initialization was performed once, the driver may enter the uninitialized state if another device on the same channel subsequently calls the I^{2}C driver setting function. In such cases, call the I^{2}C driver initialization function once again before calling the start function.

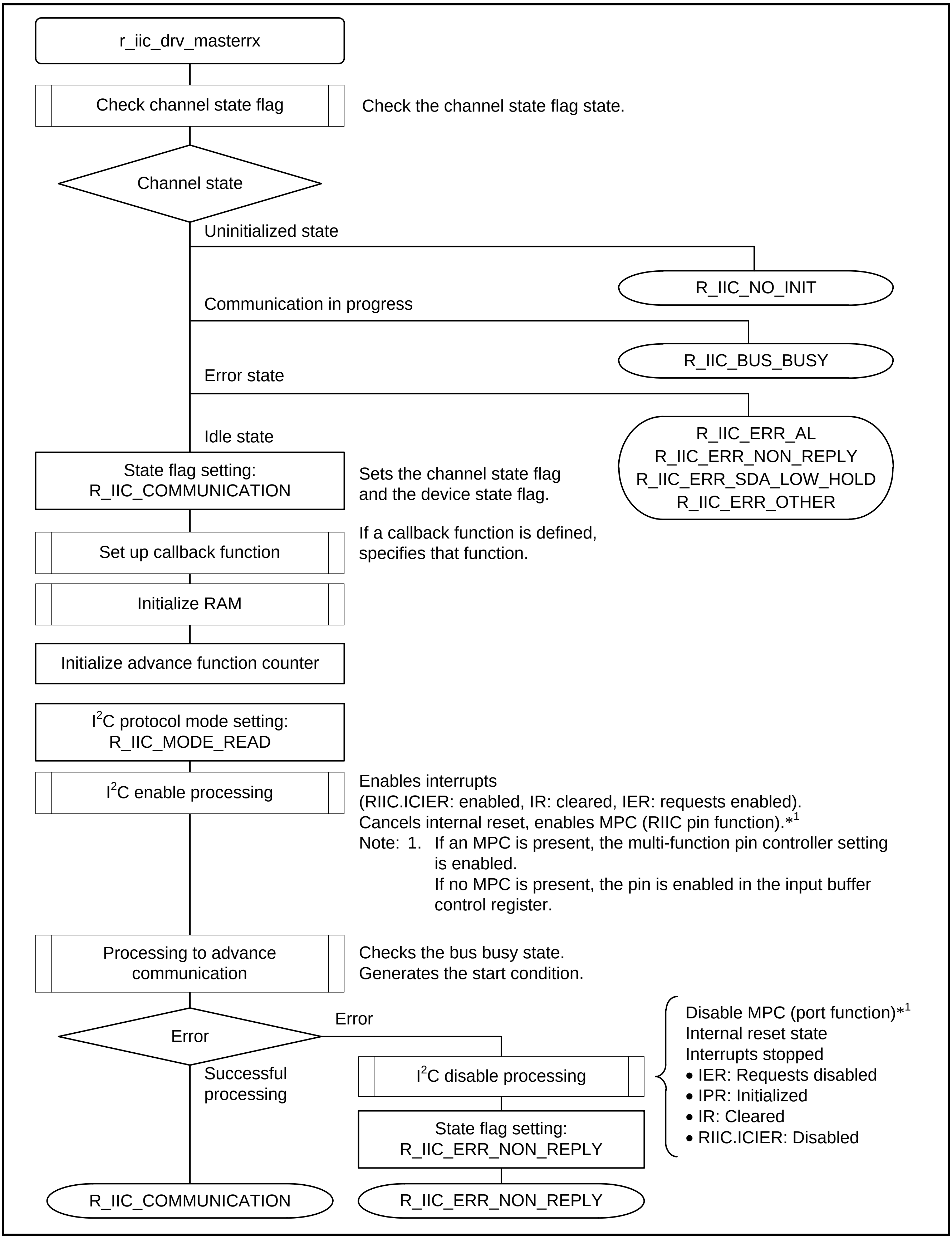


Figure 6.26 Master Reception Start Function Overview Flowchart

6.18.5 Master Composite Start Function

R_IIC_Drv_MasterTRx	
Outline	Master composite start function
Header	r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h
Declaration	error_t R_IIC_Drv_MasterTRx (r_iic_drv_info_t *pRIic_Info)
Description	• Starts master composite communication. • The r_iic_drv_info_t I^{2}C communication information structure must be set up to perform this operation. See table 6.17 for details on that setup.
Arguments	r_iic_drv_info_t *pRIic_Info ; Pointer to I^{2}C communication information structure
Return Value	R_IIC_COMMUNICATION Master composite communication was started. The channel state flag and device state flag are set to R_IIC_COMMUNICATION. → Call the advance function to terminate communication. R_IIC_NO_INIT Initialization was not performed.*1 The channel state flag and device state flag are not changed. → Call the initialization function and assure its processing has completed. R_IIC_LOCK_FUNC The processing was not performed because another API operation was being performed. The channel state flag and device state flag are not changed. → Call the function after processing of the other API finishes. R_IIC_BUS_BUSY Communication is in progress. It was not possible to start master composite communication. The channel state flag and device state flag are not changed. → Call the advance function to terminate communication. R_IIC_ERR_PARAM A parameter error was detected. The arguments were not set up. The channel state flag and device state flag are not changed. → Set up the arguments as required by this function. R_IIC_ERR_AL Arbitration was lost. The channel state flag and device state flag are not changed. → See section 7.3, Recovery Processing Example, and perform that recovery processing. R_IIC_ERR_NON_REPLY Either the bus was not released or it was not possible to detect the start condition. The channel state flag and device state flag are set to R_IIC_ERR_NON_REPLY. → See section 7.3, Recovery Processing Example, and perform that recovery processing. R_IIC_ERR_SDA_LOW_HOLD SDA is in the state where it has not recovered from the low-level hold state. The channel state flag and device state flag are not changed. → Check the system states, including whether a slave device is holding SDA low and whether a low-level signal has not been output from the master device. R_IIC_ERR_OTHER Some other error occurred. The channel state flag and device state flag are set to R_IIC_ERR_OTHER. If an error had already occurred, no processing is performed. The channel state flag and device state flag are not changed. → Check the following items. — Check that the I^{2}C communication information structure is set up correctly.

Remarks

- At the point this function returns, I^{2}C communication has not completed. The advance function must be called to terminate I^{2}C communication.
- The communication state after calling the start function can be checked with the return value from the advance function.

Note: 1. Even if initialization was performed once, the driver may enter the uninitialized state if another device on the same channel subsequently calls the I^{2}C driver setting function. In such cases, call the I^{2}C driver initialization function once again before calling the start function.

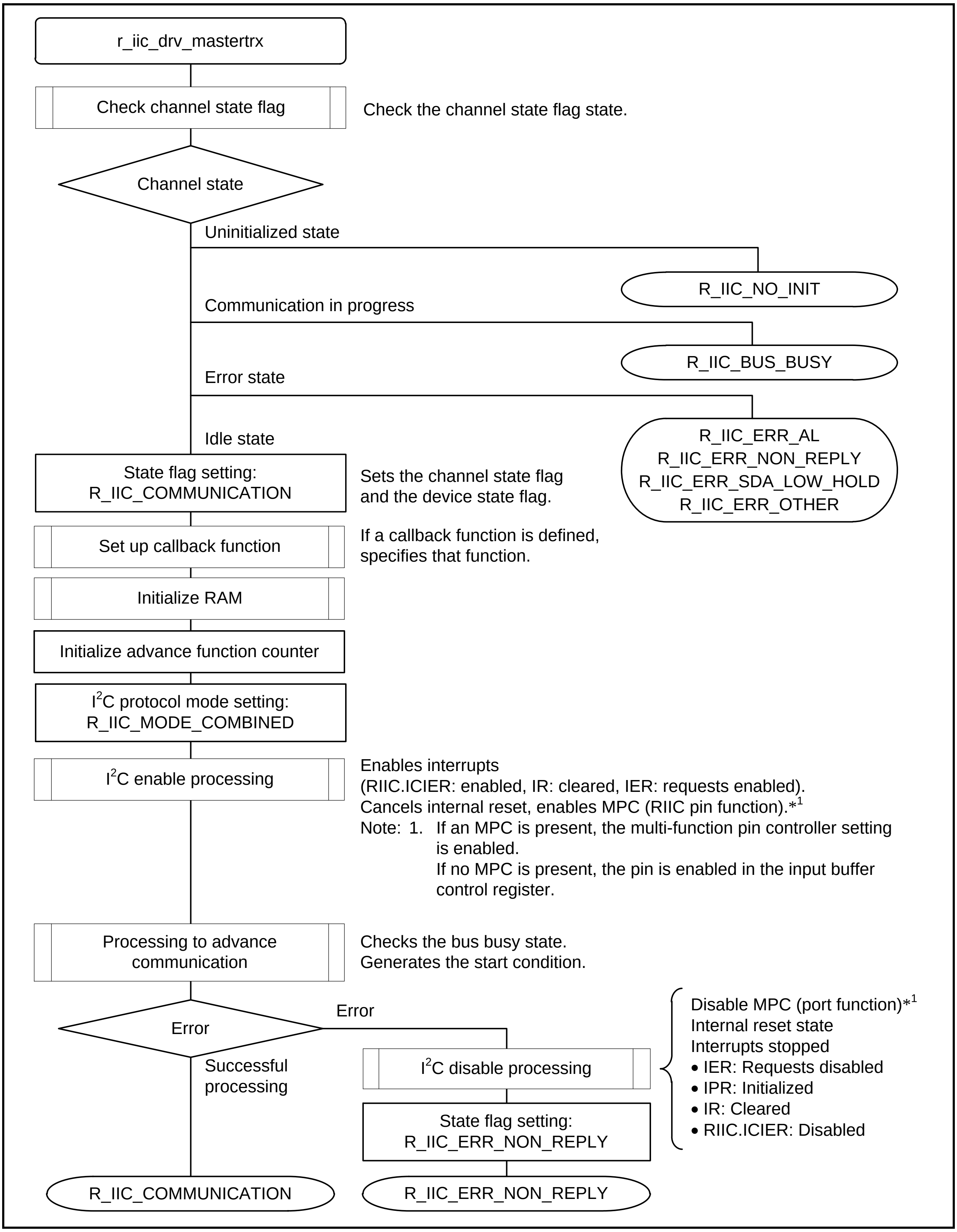


Figure 6.27 Master Composite Start Function Overview Flowchart

6.18.6 Advance Function

R_IIC_Drv_Advance

Outline Advance function

Header r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h

Declaration error_t R_IIC_Drv_Advance (r_iic_drv_info_t *pRIic_Info)

Description

- Monitors the communication and performs processing to advance communication. Returns the communication state in the return value.
- It is necessary to terminate communication with the advance function to start the next communication.

Arguments r_iic_drv_info_t *pRIic_Info ; Pointer to I^{2}C communication information structure

Return Value

R_IIC_COMMUNICATION

Communication is in progress. The channel state flag and device state flag are not changed.

→ Call the advance function to terminate communication.

R_IIC_FINISH

All communication completed successfully. The channel state flag and device state flag are set to R_IIC_FINISH.

Performs no processing if communication had already terminated. The channel state flag and device state flag are not changed.

→ Communication is now possible by calling the start function.

R_IIC_NACK

NACK was detected. A stop condition was generated and communication terminated. The channel state flag and device state flag are set to R_IIC_NACK.

Performs no processing if communication had already terminated. The channel state flag and device state flag are not changed.

→ Communication is now possible by calling the start function.

R_IIC_NO_INIT

Initialization was not performed. The channel state flag and device state flag are not changed.

→ Call the initialization function and assure its processing has completed.

R_IIC_IDLE

The system is in the idle state. The channel state flag and device state flag are not changed.

→ Communication is now possible by calling the start function.

R_IIC_LOCK_FUNC

The processing was not performed because another API operation was being performed. The channel state flag and device state flag are not changed.

→ Call the function after processing of the other API finishes.

R_IIC_BUS_BUSY

The requested processing was not performed because another device was communicating on the same channel. The channel state flag and device state flag are not changed.

→ Terminate the communication with the other device.

R_IIC_ERR_PARAM

A parameter error was detected. The arguments were not set up. The channel state flag and device state flag are not changed.

If an error had already occurred, no processing is performed. The channel state flag and device state flag are not changed.

→ Set up the arguments as required by this function.

R_IIC_ERR_AL

Arbitration was lost. The channel state flag and device state flag are set to R_IIC_ERR_AL.

If an error had already occurred, no processing is performed. The channel state flag and device state flag are not changed.

→ See section 7.3, Recovery Processing Example, and perform that recovery processing.

R_IIC_ERR_NON_REPLY

The following occurred. The channel state flag and device state flag are set to R_IIC_ERR_NON_REPLY.

- The number of calling the advance function exceeded the limit.
- Although stop condition generation processing was performed, a stop condition was not detected within a fixed period.

If an error had already occurred, no processing is performed. The channel state flag and device state flag are not changed.

→ SDA or SCL may have been held low due to noise or some other problem. See section 7.3, Recovery Processing Example, and perform that recovery processing.

R_IIC_ERR_SDA_LOW_HOLD

SDA is in the state where it has not recovered from the low-level hold state. The channel state flag and device state flag are not changed.

→ Check the system states, including whether a slave device is holding SDA low and whether a low-level signal has not been output from the master device.

R_IIC_ERR_OTHER

Some other error occurred. The channel state flag and device state flag are set to R_IIC_ERR_OTHER.

If an error had already occurred, no processing is performed. The channel state flag and device state flag are not changed.

→ Check the following items.

- Check that the I²C communication information structure is set up correctly.

Remarks

- This function checks the parameters.
- If the event flag (g_iic_Event[]) is set, the following processing is performed.
  The advance function counter (g_iic_ReplyCnt[]) is initialized.
  Communication advance processing is performed.
  If the processing proceeded successfully, the function checks whether all communication completed. When all communication has completed, the channel state flag is set to R_IIC_FINISH.
- If the event flag (g_iic_Event[]) is not set, the following processing is performed.
  The advance function counter (g_iic_ReplyCnt[]) is decremented.
  If the advance function counter is 0, the return value is set to R_IIC_ERR_NON_REPLY.

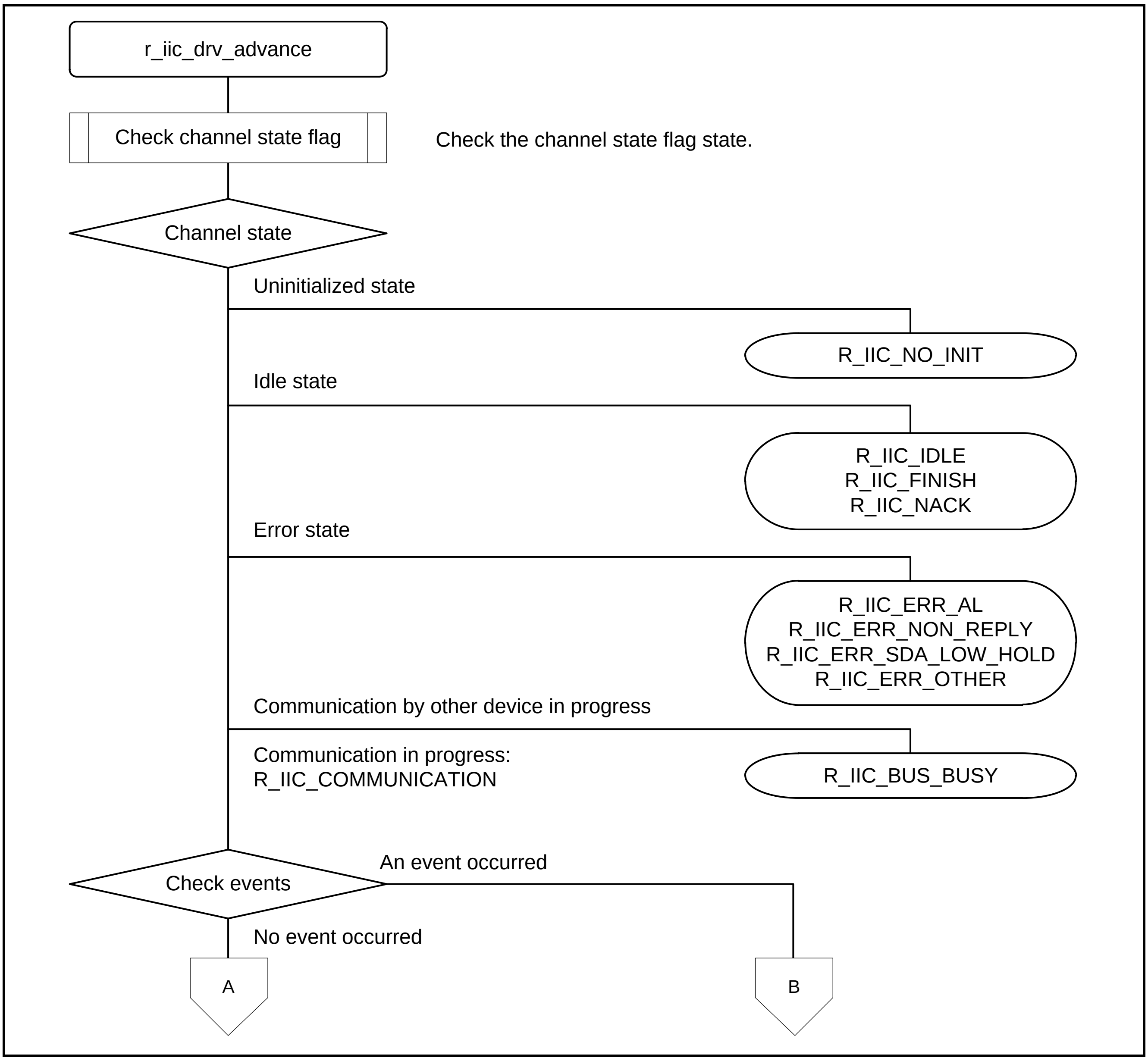
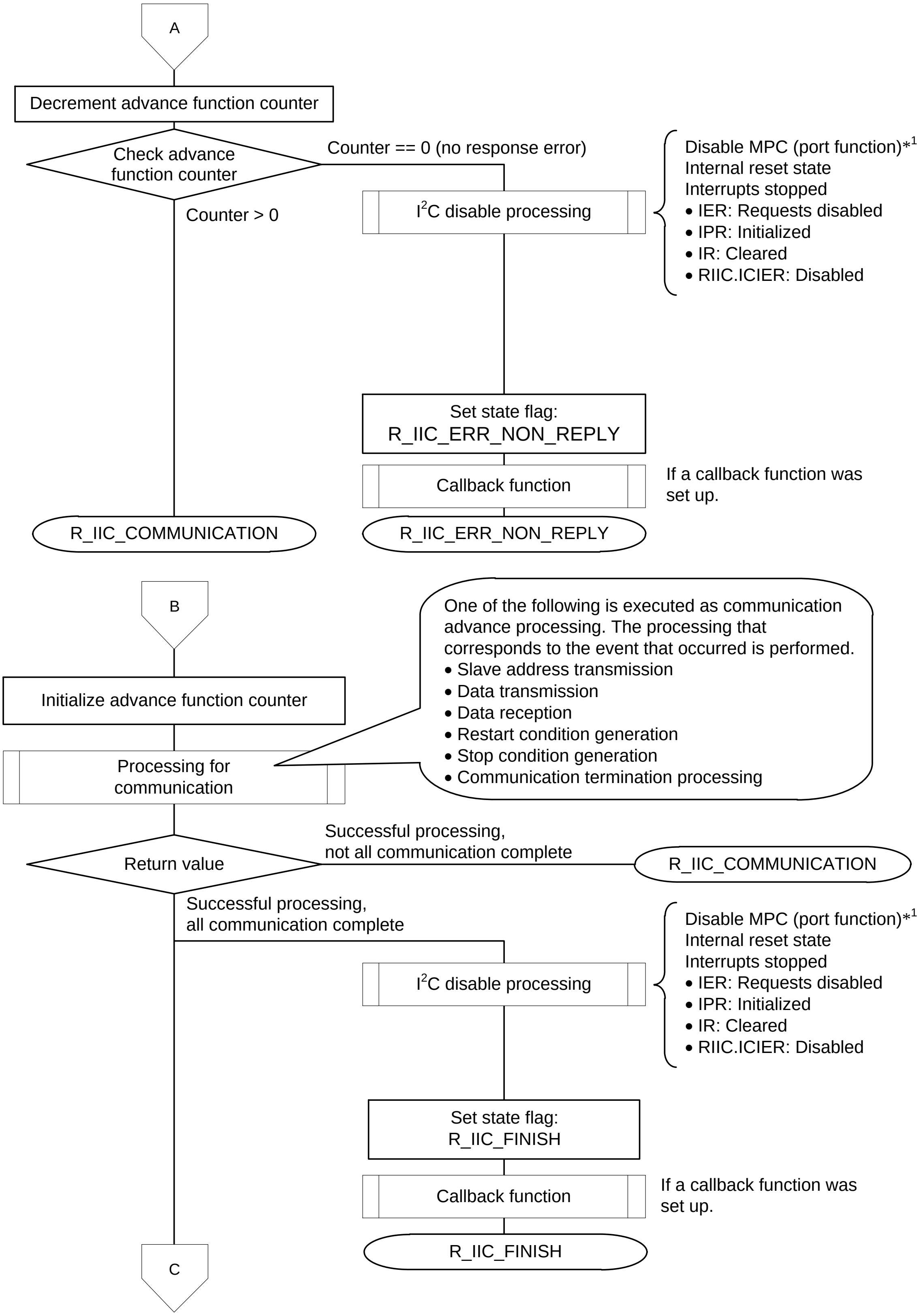
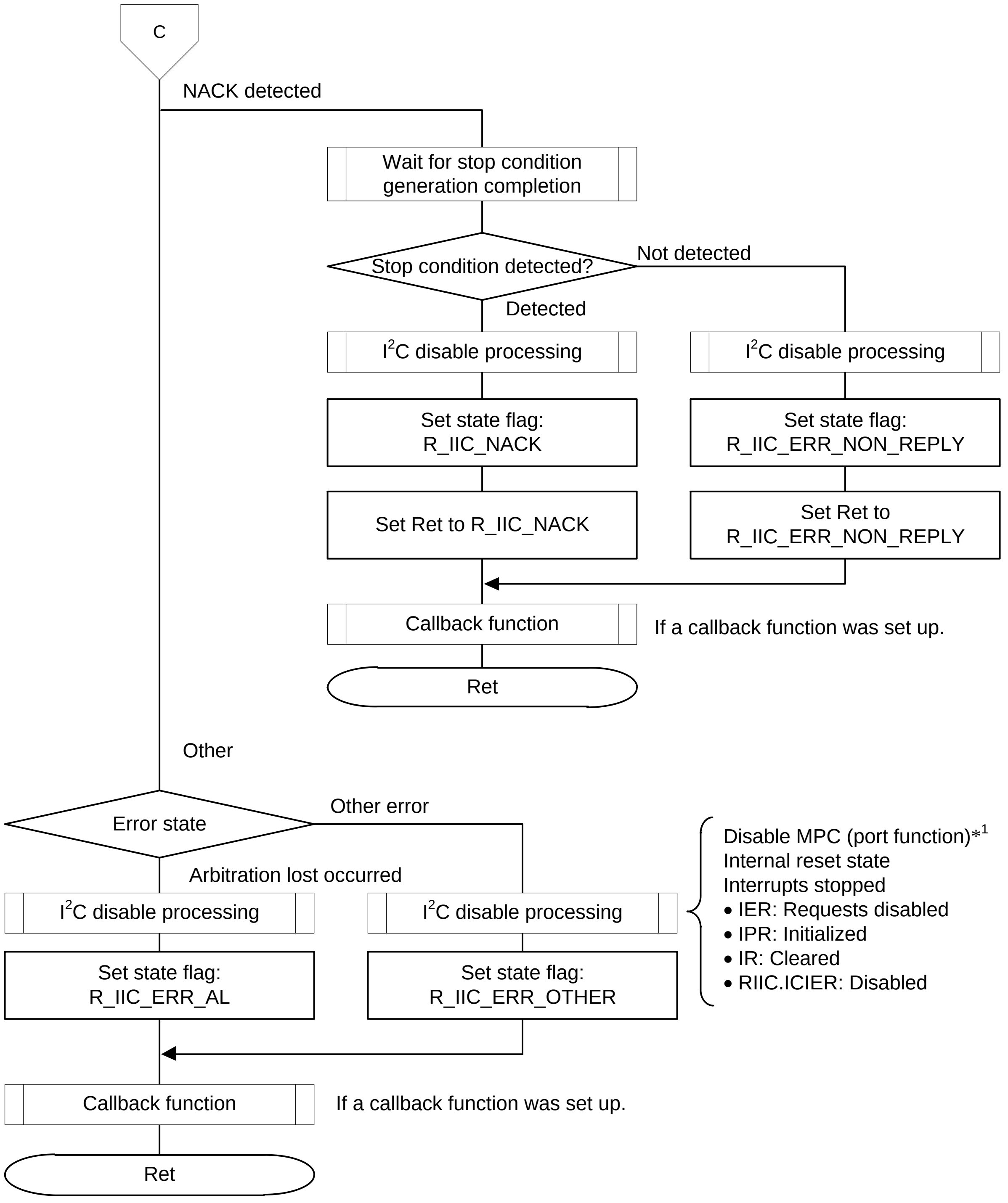


Figure 6.28 Advanced Function Overview Flowchart (1/3)



Note: 1. If an MPC is present, the multi-function pin controller setting is disabled.
If no MPC is present, the pin is disabled in the input buffer control register.

Figure 6.29 Advanced Function Overview Flowchart (2/3)



Note: 1. If an MPC is present, the multi-function pin controller setting is disabled.
If no MPC is present, the pin is disabled in the input buffer control register.

Figure 6.30 Advanced Function Overview Flowchart (3/3)

6.18.7 SCL Pseudo Clock Generation Function

R_IIC_Drv_GenClk

Outline SCL pseudo clock generation function

Header r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h

Declaration error_t R_IIC_Drv_GenClk (r_iic_drv_info_t *pRIic_Info, uint8_t ClkCnt)

Description

- This function generates an SCL pseudo clock. If a synchronization discrepancy occurs between the master and slave due to noise or other problem and SDA is held at the low level, this function can correct the internal state of the slave.
- Do not use this function in normal states. Use of this function during normal operation can result in communication problems.
- The following must be set up to use this function.
  The ChNo member of the r_iic_drv_info_t structure; The channel number used
  The clock count ClkCnt; 01h to FFh

Arguments

r_iic_drv_info_t *pRIic_Info	; Pointer to I^{2}C communication information structure
uint8_t ClkCnt	; SCL clock count

Return Value

R_IIC_NO_INIT

The SDA line has gone to the high level, correction of the internal state of the slave device completed, and the system is in the uninitialized state. The channel state flag and device state flag are set to R_IIC_NO_INIT.

→ Perform the following operations to restart communication.

(1) Call the initialization function
(2) Call master transmission with pattern 4*[1]
(3) Terminate communication by calling the advance function.

R_IIC_LOCK_FUNC

The processing was not performed because another API operation was being performed. The channel state flag and device state flag are not changed.

→ Call the function after processing of the other API finishes.

R_IIC_ERR_PARAM

A parameter error was detected. The arguments were not set up. The channel state flag and device state flag are not changed.

→ Set up the arguments as required by this function.

R_IIC_ERR_SDA_LOW_HOLD

Although an SCL pseudo clock was generated, SDA remains in the low hold state. The channel state flag and device state flag are set to R_IIC_ERR_SDA_LOW_HOLD.

→ Check the system states, including whether a slave device is holding SDA low and whether a low-level signal has not been output from the master device.

R_IIC_ERR_OTHER

The clock could not be generated. The channel status flag and device status flag are set to R_IIC_ERR_OTHER.

→ The clock can be output under the following conditions.

- Bus free state (ICCR2.BBSY flag = 0) or master mode (ICCR2.MST bit = 1 and BBSY flag =1).
- The SCL line of the communication device is not being held low.

Remarks

- If SDA is at the low level when SDA is set to the high-impedance state, the bus will be seen as not having been released.
- When SDA is low, the SCL pin is switched to port output, and a clock (low->high) is input to the bus until SDA goes high.
- An error is returned if SDA remains low when the set number of clock cycles have been generated.
- Since it is common for communication units to consist of 9 clock cycles, we recommend setting the number of clock cycles to at least 9 cycles.

Note: 1. In master transmission (pattern 4) the stop condition is generated after generation of the start condition. This processing is performed to ensure that the bus is free.

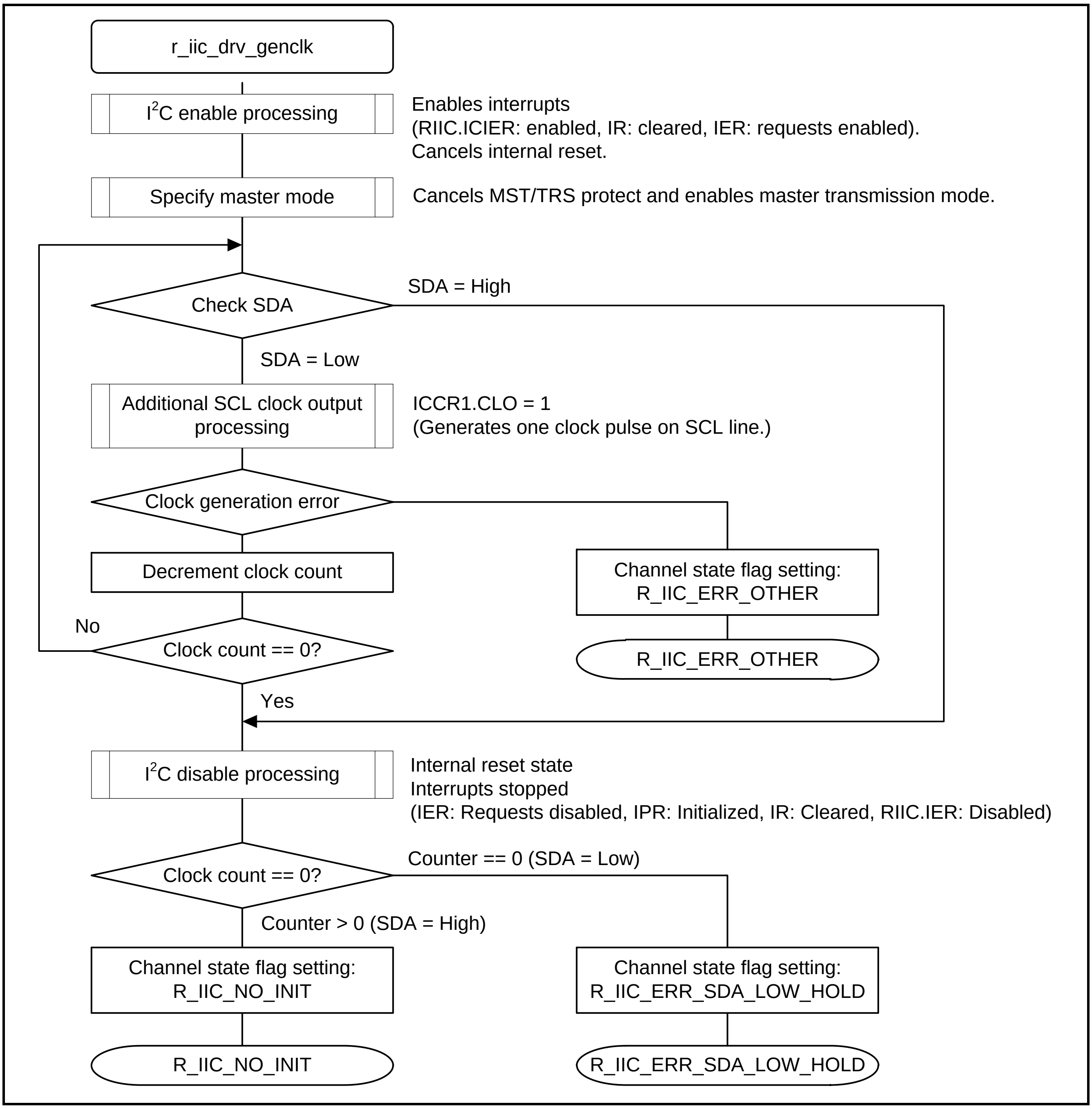


Figure 6.31 SCL Pseudo-Clock Generation Function Overview Flowchart

6.18.8 I^{2}C Driver Reset Function

R_IIC_Drv_Reset

Outline I^{2}C driver reset function

Header r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h

Declaration error_t R_IIC_Drv_Reset(r_iic_drv_info_t *pRIic_Info)

Description

- Resets the I^{2}C driver for the corresponding channel.
- Stops the RIIC and performs an internal reset by setting ICCR1.ICE to 1 and IICRST to 1.*1
- If this function is called while communication is in progress, it forcibly stops that communication.
- The following must be set up to use this function.
  The ChNo member of the r_iic_drv_info_t structure; The channel number used

Arguments r_iic_drv_info_t *pRIic_Info ; Pointer to I^{2}C communication information structure

Return Value

R_IIC_NO_INIT

An internal reset was performed and the RIIC goes to the uninitialized state. The channel state flag and device state flag are set to R_IIC_NO_INIT.

→ Perform the following operations to restart communication.

(1) Call the initialization function
(2) Call master transmission with pattern 4*2
(3) Terminate communication by calling the advance function.

R_IIC_LOCK_FUNC

The processing was not performed because another API operation was being performed. The channel state flag and device state flag are not changed.

→ Call the function after processing of the other API finishes.

R_IIC_ERR_PARAM

A parameter error was detected. The arguments were not set up. The channel state flag and device state flag are not changed.

→ Set up the arguments as required by this function.

R_IIC_ERR_OTHER

An error other than the above occurred. The channel status flag and device status flag are set to R_IIC_ERR_OTHER.

No processing takes place if an error has already occurred. The channel state flag and device state flag are not changed.

→ Check the following items.

— Check that the I^{2}C communication information structure is set up correctly.

Remarks

- To restart communication, it is also necessary to call the I^{2}C driver initialization function.
- If the RIIC is forcibly stopped during communication, the results of that communication are not guaranteed.

Notes: 1. Registers and bits affected by an internal reset
SCLO and SDAO bits in I^{2}C bus control register 1 (ICCR1)
ST bit in I^{2}C bus control register 2 (ICCR2)
BC[2:0] bits in I^{2}C bus mode register 1 (ICMR1)
I^{2}C bus status register 1 (ICSR1)
I^{2}C bus status register 2 (ICSR2)
I^{2}C bus shift register (ICDRS)

2. In master transmission (pattern 4) the stop condition is generated after generation of the start condition. This processing is performed to ensure that the bus is free.

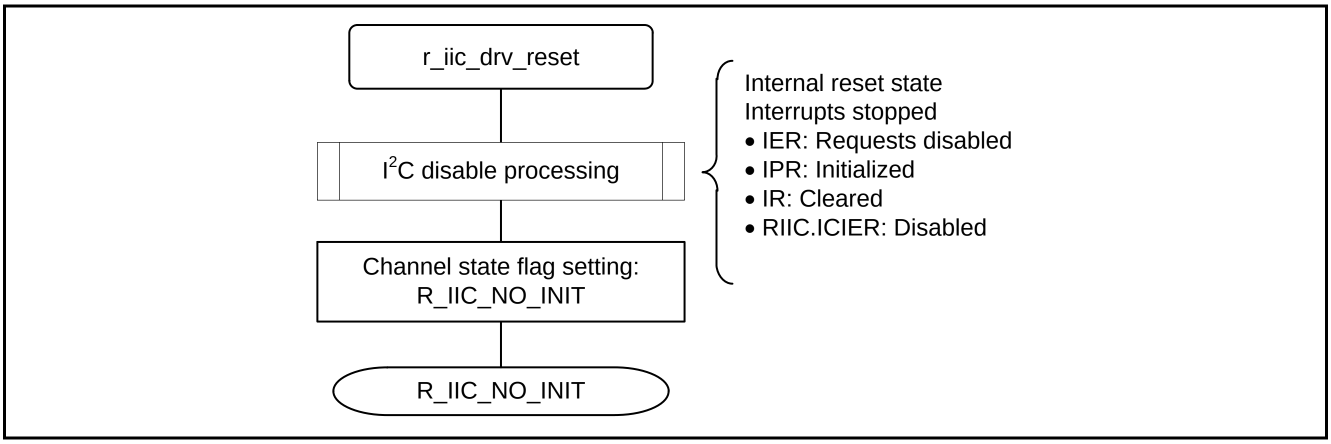


Figure 6.32 I²C Driver Reset Function Overview Flowchart

7. Application Example

7.1 r_iic_drv_api.h

This section presents and examples of settings for actual use.

The section in each file that need to be set are marked with the comment "/**SET**".

(1) Selecting the RIIC Channel Used

Specify the I^{2}C channel used. The amount of ROM used can be minimized by commenting out the unused channels.

In the example below, channels 0 and 1 are used.

```
/*------------------------------------------------------------------------*/
/*   Select channels to enable.                                           */
/*------------------------------------------------------------------------*/
#define RIIC0_ENABLE
#define RIIC1_ENABLE
```

(2) Defining the Maximum Number of Channels Used

Set this item to the largest channel number used plus one.

In the example below, channels 0 and 1 are used. Since the largest channel number used here is 1, this item is set to 2.

```
/*------------------------------------------------------------------------*/
/*   Define channel No.(max) + 1.                                         */
/*------------------------------------------------------------------------*/
#define MAX_IIC_CH_NUM          (uint8_t)(2)
```

(3) Definition when Calling the Advance Function from the RIIC Interrupt Handler

Make the following definition if the advance function will be called by the RIIC interrupt handler.

```
/*------------------------------------------------------------------------*/
/*   Define to use an advance processing by RIIC interrupt handler.       */
/*------------------------------------------------------------------------*/
#define CALL_ADVANCE_INTERRUPT
```

(4) Counter Definitions

These are the counter values for various software loops. As such, the loop times will change with the system clock used. These setting values should be reviewed as necessary.

```
/*------------------------------------------------------------------------*/
/*   Define counter.                                                      */
/*------------------------------------------------------------------------*/
#define REPLY_CNT        (uint32_t)(100000) /* Counter of non-reply errors   */
#define STOP_COND_WAIT   (uint16_t)(1000)
                            /* Counter of waiting stop condition generation */
#define BUSCHK_CNT       (uint16_t)(1000) /* Counter of checking bus busy    */
#define SDACHK_CNT       (uint16_t)(1000) /* Counter of checking SDA level   */
#define GEN_SCLCLK_WAIT   (uint16_t)(1000)
                            /* Counter of waiting SCL clock setting         */
```

7.2 r_iic_drv_sfr.h

A file with a filename of the form r_iic_drv_sfr.hXXX has been created for each microcontroller. One of these must be renamed to r_iic_drv_sfr.h and used. If there is no such file for the microcontroller used, the user must refer to these files and create an appropriate r_iic_drv_sfr.h file.

This section presents and examples of settings for actual use.

The section in each file that need to be set are marked with the comment "/**SET**/".

(1) Defining the Transfer Clock

Define the transfer clock by making the settings listed below. These values must be set for each channel used. See the RX Family User’s Manual: Hardware for details on the setting procedure.

- Internal reference clock select bits (CKS[2:0]) in I^{2}C bus mode register 1 (ICMR1)
- I^{2}C bus bit rate low-level register (ICBRL)
- I^{2}C bus bit rate high-level register (ICBRH)

The maximum setting is 400 kHz. However, if standard mode devices and fast mode devices are used together, the standard mode maximum setting of 100 kHz must be used. Note that it may be necessary to modify the setting value since the rise time (tR) and fall time (tF) of the SDA and SCL signals differ according to the pull-up resistance and the wiring capacitance.

Sample settings for channels 0 and 1 are shown below.

```
/*--------------------------------------------------------------------------*/
/*   Define frequency as iic channel. (Please add a channel as needed.)     */
/*--------------------------------------------------------------------------*/
/* The I2C transfer rate is calculated using the following expression.      */
/* Transfer rate = 1 / {[(ICBRH + 1) + (ICBRL + 1)] / (PCLK*Division ratio)
                + SCLn line rising time [tr] + SCLn line falling time [tf]} */
/* Note1:Division ratio sets it by ICMR1.CKS[2:0].                          */

/* Freq = 400KHz at main system clock = 48MHz */
#define R_IIC_CH0_LCLK  (uint8_t)(0xED) /* Channel 0 ICBRL register setting */
#define R_IIC_CH0_HCLK  (uint8_t)(0xE6) /* Channel 0 ICBRH register setting */
#define R_IIC_CH1_LCLK  (uint8_t)(0xED) /* Channel 1 ICBRL register setting */
#define R_IIC_CH1_HCLK  (uint8_t)(0xE6) /* Channel 1 ICBRH register setting */

/* Sets ICMR1 register.*/
#define R_IIC_CH0_ICMR1_INIT    (uint8_t)(0x28)
                                        /* Channel 0 ICMR1 register setting */
#define R_IIC_CH1_ICMR1_INIT    (uint8_t)(0x28)
                                        /* Channel 1 ICMR1 register setting */
```

(2) Defining Port Numbers

When using channel 0 on the RX210 or RX21A, define the port numbers of the pins to be used.

Sample settings for port 12 (SCL0) and port 13 (SDA0) of channel 0 are shown below.

```
/*--------------------------------------------------------------------------*/
/*   Define channel register.                                               */
/*--------------------------------------------------------------------------*/
#ifdef RIIC0_ENABLE

/* Define port registers */
#define R_IIC_PDR_SCL0     PORT1.PDR.BIT.B2
                                 /* SCL0 Port direction register             */
#define R_IIC_PDR_SDA0     PORT1.PDR.BIT.B3
                                 /* SDA0 Port direction register             */
#define R_IIC_PODR_SCL0    PORT1.PODR.BIT.B2
                                 /* SCL0 Port output data register           */
#define R_IIC_PODR_SDA0    PORT1.PIDR.BIT.B3
                                 /* SDA0 Port output data register           */
#define R_IIC_PIDR_SCL0    PORT1.PODR.BIT.B2
                                 /* SCL0 Port input data register            */
#define R_IIC_PIDR_SDA0    PORT1.PIDR.BIT.B3
                                 /* SDA0 Port input data register            */
#define R_IIC_PMR_SCL0     PORT1.PMR.BIT.B2  /* SCL0 Port mode register      */
#define R_IIC_PMR_SDA0     PORT1.PMR.BIT.B3  /* SDA0 Port mode register      */
#define R_IIC_DSCR_SCL0    PORT1.DSCR.BIT.B2
                                 /* SCL0 Drive capacity control register     */
#define R_IIC_DSCR_SDA0    PORT1.DSCR.BIT.B3
                                 /* SDA0 Drive capacity control register     */
#define R_IIC_PCR_SCL0     PORT1.PCR.BIT.B2
                                 /* SCL0 Pull-up resistor control register   */
#define R_IIC_PCR_SDA0     PORT1.PCR.BIT.B3
                                 /* SDA0 Pull-up resistor control register   */
```

(3) Multi-Function Pin Controller (MPC) Definitions

When using channel 0 on the RX210 or RX21A, define the multi-function pin controller (MPC) register numbers of the pins to be used.

Sample settings for port 12 (SCL0) and port 13 (SDA0) of channel 0 are shown below.

```
/* Define Pin function control registers */
#define R_IIC_MPC_SCL0  MPC.P12PFS.BYTE
                                 /* SCL0 Pin function control register       */
#define R_IIC_MPC_SDA0  MPC.P13PFS.BYTE
                                 /* SDA0 Pin function control register       */
```

(4) Defining the RIIC Interrupt Priorities

Define the interrupt priorities of the RIIC channel to be used by making the appropriate settings in the interrupt source priority register (IPR). These values must be set for each channel used.

Sample settings for defining the priorities of the RIIC interrupts as level 2 are shown below.

```
/* Sets interrupt source priority initialization. */
#define R_IIC_IPR_CH0_EEI_INIT    (uint8_t)(0x02)
                           /* EEIx interrupt source priority initialization*/
#define R_IIC_IPR_CH0_RXI_INIT    (uint8_t)(0x02)
                           /* RXIx interrupt source priority initialization*/
#define R_IIC_IPR_CH0_TXI_INIT    (uint8_t)(0x02)
                           /* TXIx interrupt source priority initialization*/
#define R_IIC_IPR_CH0_TEI_INIT    (uint8_t)(0x02)
                           /* TEIx interrupt source priority initialization*/
```

7.2.1 Interrupt Handler Settings

The interrupts used in the sample code are the ICEEI, ICTEI, and ICRXI interrupts.

Sample settings are shown below for the case where the vect.h (headers of vector function) and intprg.c (vector function definitions) files generated by the integrated development environment are used and for the case where they are not used.

(1) Using the Generated Files

Define the interrupt handler functions for the channel used by r_iic_drv_int.c in the portion of intprg.c that defines the RIIC interrupts.

Sample settings for channel 0 are shown below.

```
// RIIC0 EEI0
void Excep_RIIC0_EEI0(void){ r_iic_drv_intRIIC0_EEI_isr(); }

// RIIC0 RXI0
void Excep_RIIC0_RXI0(void){ r_iic_drv_intRIIC0_RXI_isr(); }

// RIIC0 TEI0
void Excep_RIIC0_TEI0(void){ r_iic_drv_intRIIC0_TEI_isr(); }
```

(2) Not Using the Generated Files

Define #pragma interrupt as the interrupt handler function for the channel used by r_iic_drv_int.c.

Sample settings for channel 0 are shown below.

ICEEI Interrupt Definition

```
#pragma interrupt (r_iic_drv_intRIIC0_EEI_isr(vect=VECT_RIIC0_RXI0))
void r_iic_drv_intRIIC0_EEI_isr(void)
```

ICRXI Interrupt Definition

```
#pragma interrupt (r_iic_drv_intRIIC0_RXI_isr(vect=VECT_RIIC0_RXI0))
void r_iic_drv_intRIIC0_RXI_isr(void)
```

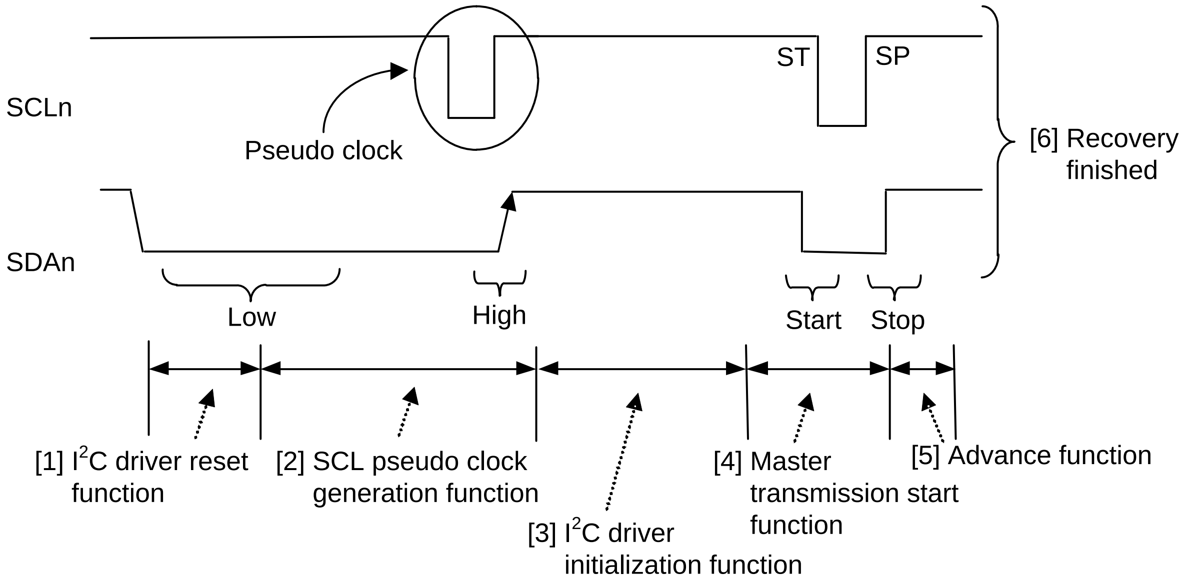
ICTEI Interrupt Definition

```
#pragma interrupt (r_iic_drv_intRIIC0_TEI_isr(vect=VECT_RIIC0_TEI0))
void r_iic_drv_intRIIC0_TEI_isr(void)
```

7.3 Recovery Processing Example

Recovery processing to return to communication when SDA or SCL is being held low is described below. Follow the steps shown below in the processing. Figure 7.1 shows an example of recovery processing by means of SCL pseudo clock generation.

Note that the RIIC enters idle mode after the recovery processing described here. From this state, communication can be initiated by calling the start function.



[1] Resets the I^2C bus internally.
[2] Generates a pseudo clock on SCL, releasing the slave device from the low-hold state.
[3] Performs initialization to enable master transmission (pattern 4).
[4] Starts master transmission (pattern 4). Generates the start condition, then generates the stop condition.
[5] Ends communication by using the advance function.
[6] The module is in the idle state. Subsequently, communication can be initiated by calling the start function.

Figure 7.1 Example of Recovery Processing Using SCL Pseudo Clock Generation

[1] I^{2}C bus driver reset function: R_IIC_Drv_Reset()

This function cancels the hold state of SDA and SCL by performing an internal reset.

After this function finishes its processing, verify that SDA and SCL are high level. If SDA or SCL remain held low after a reset, it is possible that they are being held low by the slave device or that a low signal is being output by the master device.

[2] SCL pseudo clock generation function: R_IIC_Drv_GenClk()

If SDA is being held low, this function generates a pseudo clock on SCL to end the internal processing of the slave device so that SDA goes high.

If the return value is R_IIC_SDA_HIGH, SDA is high level and the low-hold state was canceled. The return value is R_IIC_ERR_SDA_LOW_HOLD if it was not possible to release SDA from the low-hold state. In this case it is necessary to reassess the state of the system.

[3] I^{2}C driver initialization: R_IIC_Drv_Init()

If the preceding processing has released SDA and SCL from the low-hold state, call the I^{2}C driver initialization function.

[4] Bus release processing: R_IIC_Drv_MasterTx() pattern 4

Releases the bus by generating the stop condition. In the sample code, master transmission pattern 4 generates the start condition and then generates the stop condition.

Make settings for pattern 4, then call the master transmission start function.

[5] Advance function: R_IIC_Drv_Advance()

Call the advance function to finish the processing started in item [4].

The RIIC enters the idle state after a successful end. From this state, communication can be initiated by calling the start function.

7.4 Notes on Using RIIC Interrupt Handler to Call Advance Function

Make sure to ensure the conditions listed below when using the RIIC interrupt handler to call the advance function.

(1) Enabling of #define CALL_ADVANCE_INTERRUPT

As described in item (3) of 7.1, define CALL_ADVANCE_INTERRUPT.

(2) Defining of I^{2}C Communication Information Structure

Define the following global I^{2}C communication information structure in the main processing routine, as defined in r_iic_drv_int.h. This definition must be made for each channel used.

```
#ifdef CALL_ADVANCE_INTERRUPT
    #ifdef RIIC0_ENABLE
extern r_iic_drv_info_t   g_iic_Info_ch0;   /* Channel 0 IIC driver
information*/
    #endif /* #ifdef RIIC0_ENABLE */
    #ifdef RIIC1_ENABLE
extern r_iic_drv_info_t   g_iic_Info_ch1;   /* Channel 1 IIC driver
information*/
    #endif /* #ifdef RIIC1_ENABLE */
    #ifdef RIIC2_ENABLE
extern r_iic_drv_info_t   g_iic_Info_ch2;   /* Channel 2 IIC driver
information*/
    #endif /* #ifdef RIIC2_ENABLE */
    #ifdef RIIC3_ENABLE
extern r_iic_drv_info_t   g_iic_Info_ch3;   /* Channel 3 IIC driver
information*/
    #endif /* #ifdef RIIC3_ENABLE */
#endif  /*#ifdef CALL_ADVANCE_INTERRUPT*/
```

(3) Addition of RIIC Interrupt Disable/Enable Processing when Calling Start Function

When using the RIIC interrupt handler to call the advance function, add processing on the user side to disable and enable RIIC interrupts before and after calls to the various start functions.

If during the above interval an RIIC interrupt occurs and the RIIC interrupt handler calls the advance function, multiple API calls will overlap and processing will end before the advance function can run. This will prevent subsequent communication from occurring.

(4) Method of Determining Completion of Communication

To confirm that communication has finished, specify a callback function to set a flag, etc. The callback function is called when either a successful end or an error end occurs.

The callback function should be created by the user and specified in the CallBackFunc member of the I^{2}C communication information structure.

(5) Method of Determining Successful End and Error End

After communication ends, whether a successful end and error end occurred can be confirmed by reading the channel status flag (g_iic_ChStatus[]).

(6) Disabling of Calls to API Functions Other Than Calls to Advance Function by RIIC Interrupt Handler During Communication

When using the RIIC interrupt handler to call the advance function, do not make calls to API functions other than calls to the advance function by the RIIC interrupt handler while communication is in progress.

If an API function is called while communication is in progress and an RIIC interrupt occurs while the API function is running, multiple API calls will overlap when the RIIC interrupt handler calls the advance function, causing processing to end before the advance function can run. This will prevent subsequent communication from occurring.

8. Usage Notes

8.1 Notes on Embedding

8.1.1 Include File

Include the following header files when embedding this sample code in an application.

- r_iic_drv_api.h
- r_iic_drv_sub.h
- r_iic_drv_sfr.h
- r_iic_drv_int.h

8.2 Notes on Initialization

When performing initialization for the first time after system startup, set the channel status flag g_iic_ChStatus[] to R _IIC_NO_INIT for all channels to be used. Also, set the device status flag *(pRIic_Info.pDevStatus) to R_IIC_NO_INIT for all slave devices to be used.

After setting both flags, set the structure information for the slave devices to be used in the I^{2}C driver initialization function, then call the I^{2}C driver initialization function. Complete initialization of all slave devices as the first step before proceeding.

After this, making settings to the channel status flags and device status flags is prohibited as this is handled by the sample code.

8.3 Notes on the Channel State Flag and Device State Flag

This sample code maintains the consistency of the communication state using the channel state flag and device state flag. Communication operation is not guaranteed if these flags are modified after first initialization.

8.4 Operation Verification Program

The operation verification program supplied with the sample code writes to and reads from EEPROM.

8.5 Example of Embedding

Refer to the operation verification program sample_background.c for the method of embedding to be used when calling the advance function from the RIIC interrupt handler. Also make sure to follow the guidelines contained in 7.4, Notes on Using RIIC Interrupt Handler to Call Advance Function.

For the method of embedding to be used when calling the advance function from the main processing routine, refer to operation verification program sample_foreground.c.

8.6 Control Methods for Multiple Slave Devices on the Same Channel

Use the following procedure to control multiple slave devices on the same channel.

The processing in the item (1) below can prevent communication from being performed with devices in the not communicating state.

(1) Verify that the device state flag in the I^{2}C communication information structure for the slave device that is the object of the advance function call is “R_IIC_COMMUNICATION”.
(2) Call the advance function.
(3) Repeat steps (1) and (2) until communication completes.
(4) Communication has completed. After this, communication is possible by calling a start function.

The advance function moves forward the processing of the slave device while communication is in progress, without identifying the specific slave device. Therefore, the processing of a slave device can be moved forward while communication is in progress even if the value of the device status flag is other than R_IIC_COMMUNICATION (communication in progress).

8.7 Transfer Rate Setting

The transfer rate must be set for each channel. Transfer rates up to a maximum of 400 kHz can be set.

Note, however, that if standard mode devices and fast mode devices are used together, the standard mode maximum rate of 100 kHz must be set. Set the transfer rate using R_IIC_CHx_LCLK, R_IIC_CHx_LCLK, and R_IIC_CHx_ICMR1_INIT (where x is the channel number) defined in table 6.14.

8.8 Notes On Setting The #define Definitions of RIICx_ENABLE and MAX_IIC_CH_NUM

This section described the settings for the case where only channel 2 will be used.

Enable only the definition of RIIC2_ENABLE for the RIICx_ENABLE #define definitions. This masks out the source code for channel 0 and channel 1.

```
/*------------------------------------------------------------------------------*/
/*   Select channels to enable.                                                 */
/*------------------------------------------------------------------------------*/
/* #define RIIC0_ENABLE */
/* #define RIIC1_ENABLE */
#define RIIC2_ENABLE
```

Set the #define definition of MAX_IIC_CH_NUM to 3. Note that although the number of channels used is 1, the value set here must be the largest channel number used plus one.

```
/*------------------------------------------------------------------------------*/
/*   Define channel No.(max) + 1.                                               */
/*------------------------------------------------------------------------------*/
#define MAX_IIC_CH_NUM        (uint8_t)(3)
```

8.9 Port Pins Assigned as RIIC Pins

The port pins on each microcontroller assigned as RIIC pins are listed below.

Table 8.1 Port Pins Assigned as RIIC Pins

MCU	Pin Count	Channel Count	RIIC0		RIIC1		RIIC2		RIIC3	
			SCL0	SDA0	SCL1	SDA1	SCL2	SDA2	SCL3	SDA3
RX62N	176	2	P12	P13	P21	P20	—	—	—	—
	145	2	P12	P13	P21	P20	—	—	—	—
	100	1	P12	P13	—	—	—	—	—	—
	85	2	P12	P13	P21	P20	—	—	—	—
RX63N/ RX631	177,176	4	P12	P13	P21	P20	P16	P17	PC0	PC1
	145,144	4	P12	P13	P21	P20	P16	P17	PC0	PC1
	100	2	P12	P13	—	—	P16	P17	—	—
	64	1	—	—	—	—	P16	P17	—	—
	48	1	—	—	—	—	P16	P17	—	—
RX63T	144	2	PB1	PB2	P25	P26	—	—	—	—
	120	2	PB1	PB2	P25	P26	—	—	—	—
	112	1	PB1	PB2	—	—	—	—	—	—
	100	1	PB1	PB2	—	—	—	—	—	—
	64	1	PB1	PB2	—	—	—	—	—	—
	48	1	PB1	PB2	—	—	—	—	—	—
RX210	145,144	1	P12, P16	P13, P17	—	—	—	—	—	—
	100	1	P12, P16	P13, P17	—	—	—	—	—	—
	80	1	P12, P16	P13, P17	—	—	—	—	—	—
	64	1	P16	P17	—	—	—	—	—	—
	48	1	P16	P17	—	—	—	—	—	—
RX21A	100	2	P12, P16	P13, P17	P21	P20	—	—	—	—
	80	2	P12, P16	P13, P17	P21	P20	—	—	—	—
	64	1	P16	P17	—	—	—	—	—	—

8.10 Microcontrollers Requiring Specification of Port Pins

As shown in 8.9, either of two ports can be used for SCL and for SDA on channel 0 on the RX210 and RX21A. When this channel us used, specify which of the two ports to be used for SCL and for SDA.

8.11 NACK Detection Processing after Direct Transmission to Slave Address with Master Transmission and Master Composite Operation

During master transmission or master composite operation, if a NACK is received on the ninth bit of slave address transmission (transfer direction bit: 1 (read)), a dummy read of the I^{2}C bus receive data register (ICDRR) occurs after the stop condition generation settings.

The receive data-full flag is set to 1 even when a NACK is detected under the above conditions. The dummy read of ICDRR is performed to clear this flag.

Website and Support

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

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

REVISION HISTORY	RX600, RX200 Application Note I²C Bus Single Master Control Software Using RIIC Serial Interface

Rev.	**Date**	**Description**	
		Page	**Summary**
1.03	Jan. 30, 2015	—	First edition issued

All trademarks and registered trademarks are the property of their respective owners.
I²C-bus is a trademark of NXP B.V.

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU 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 an MPU or MCU 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 of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
3. Renesas Electronics does not assume any liability for infringement of 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. 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 should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The recommended 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 systems; anti-disaster systems; anti-crime systems; and safety 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 (nuclear reactor control systems, military equipment etc.). You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application for which it is not intended. Renesas Electronics shall not be in any way liable 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. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages 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 its 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 be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, 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 use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technology may 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 should not use Renesas Electronics products or technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. When exporting the Renesas Electronics products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
11. This document may not be 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, or if you have any other inquiries.

(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.

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 5.0