RX600, RX200 Series
I²C Bus Single Master Control Software
Using RIIC Serial Interface

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 .................................................................................................................. 4

2. Operation Confirmation Conditions ............................................................................. 6

3. Reference Application Note ......................................................................................... 8

4. Peripheral Functions .................................................................................................... 8

5. Hardware ........................................................................................................................ 9
   5.1 Pins Used .................................................................................................................. 9
   5.2 Reference Circuit ..................................................................................................... 9
   5.3 Controlling Multiple Slave Devices ....................................................................... 10
   5.4 Maximum Transfer Speed .................................................................................... 10

6. Software ........................................................................................................................... 11
   6.1 Software Structure ................................................................................................. 11
   6.2 Operation Overview ............................................................................................... 12
      6.2.1 Master Transmission ..................................................................................... 12
      6.2.2 Master Reception ......................................................................................... 14
      6.2.3 Master Composite ....................................................................................... 15
   6.3 Software Operation .................................................................................................. 16
   6.4 Software Operating Sequence ............................................................................... 18
   6.5 Implementation of Slave Device Control ............................................................... 20
   6.6 Communication Implementation .......................................................................... 21
      6.6.1 States During Control .................................................................................. 21
      6.6.2 Events During Control ............................................................................... 21
      6.6.3 Protocol State Transitions ......................................................................... 22
      6.6.4 Protocol State Transition Table .................................................................... 26
      6.6.5 Protocol State Transition Registered Functions ........................................... 26
      6.6.6 Processing at Protocol State Transitions ..................................................... 27
   6.7 Interrupt Generation Timing .................................................................................. 28
      6.7.1 Master Transmission ..................................................................................... 28
      6.7.2 Master Reception ......................................................................................... 29
      6.7.3 Master Composite ....................................................................................... 30
   6.8 Callback Function .................................................................................................... 31
   6.9 Relationship of Data Buffers and Transmit/Receive Data ......................................... 31
   6.10 Required Memory Sizes ........................................................................................ 32
   6.11 File Structure ......................................................................................................... 33
   6.12 Constants ................................................................................................................ 34
      6.12.1 Return Values .............................................................................................. 34
      6.12.2 Definitions ................................................................................................... 35
   6.13 Structures and Unions ............................................................................................. 36
      6.13.1 I2C Communication Information Structure ................................................ 36
      6.13.2 Internal Information Management Structure ............................................... 38
   6.14 Enumerated Types .................................................................................................. 39
   6.15 Variables ....................................................................................................................... 40
   6.16 Functions ..................................................................................................................... 41
   6.17 State Transition Diagram ....................................................................................... 42
6.17.1 Error State Definitions ................................................................. 43
6.17.2 Flag States at State Transitions .................................................... 44
6.18 Function Specifications ................................................................... 45
6.18.1 Common Processing for These Functions ......................................... 45
6.18.2 I²C Driver Initialization Function .................................................... 46
6.18.3 Master Transmission Start Function ............................................... 50
6.18.4 Master Reception Start Function ..................................................... 53
6.18.5 Master Composite Start Function .................................................... 56
6.18.6 Advance Function ......................................................................... 59
6.18.7 SCL Pseudo Clock Generation Function ......................................... 64
6.18.8 I²C Driver Reset Function ............................................................... 66

7. Application Example ......................................................................... 68
  7.1 r_iic_drv_api.h ................................................................................. 68
  7.2 r_iic_drv_sfr.h ................................................................................. 69
    7.2.1 Interrupt Handler Settings ............................................................ 71
  7.3 Recovery Processing Example .......................................................... 73
  7.4 Notes on Using RIIC Interrupt Handler to Call Advance Function ........ 75

8. Usage Notes ..................................................................................... 77
  8.1 Notes on Embedding ........................................................................ 77
    8.1.1 Include File ................................................................................ 77
  8.2 Notes on Initialization ...................................................................... 77
  8.3 Notes on the Channel State Flag and Device State Flag ....................... 77
  8.4 Operation Verification Program ....................................................... 77
  8.5 Example of Embedding .................................................................... 77
  8.6 Control Methods for Multiple Slave Devices on the Same Channel .... 78
  8.7 Transfer Rate Setting ...................................................................... 78
  8.8 Notes On Setting The #define Definitions of RIICx_ENABLE and MAX_IIC_CH_NUM .... 78
  8.9 Port Pins Assigned as RIIC Pins ....................................................... 79
  8.10 Microcontrollers Requiring Specification of Port Pins ....................... 79
  8.11 NACK Detection Processing after Direct Transmission to Slave Address with Master Transmission and Master Composite Operation ........................................ 79
1. Specifications

This sample code performs I²C bus single master control using the RX Family I²C 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 I²C bus single master device driver that uses the RX Family microcontroller as the master device using its I²C bus interface.
- This sample code implements the protocols in the I²C-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 I²C 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

<table>
<thead>
<tr>
<th>Pattern</th>
<th>ST Generation</th>
<th>Slave Address Transmission</th>
<th>First Data Transmission</th>
<th>Second Data Transmission</th>
<th>SP Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern 1</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Pattern 2</td>
<td>○</td>
<td>○</td>
<td>—</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Pattern 3</td>
<td>○</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>○</td>
</tr>
<tr>
<td>Pattern 4</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>○</td>
</tr>
</tbody>
</table>

Legend:
ST: Start condition
SP: Stop condition
Table 1.2 Peripheral Function and Its Application

<table>
<thead>
<tr>
<th>Peripheral Function</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIIC</td>
<td>I^2C bus interface</td>
</tr>
<tr>
<td></td>
<td>One channel (required)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU used for evaluation</td>
<td>RX62N Group (program ROM: 512 KB, RAM: 64 KB)</td>
</tr>
<tr>
<td>Memory used for evaluation</td>
<td>Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I2C Serial EEPROM</td>
</tr>
<tr>
<td>Operating frequency</td>
<td>ICLK: 96 MHz, PCLK: 48 MHz</td>
</tr>
<tr>
<td>Operating voltage</td>
<td>3.3 V</td>
</tr>
<tr>
<td>Integrated development environment</td>
<td>Renesas Electronics High-performance embedded Workshop Version 4.09.01.007</td>
</tr>
<tr>
<td>C compiler</td>
<td>Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 1.2.1.0)</td>
</tr>
<tr>
<td>Endian mode</td>
<td>Big endian/Little endian</td>
</tr>
<tr>
<td>Sample code version</td>
<td>Ver. 1.13</td>
</tr>
<tr>
<td>Software used</td>
<td>Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01</td>
</tr>
<tr>
<td>Board used</td>
<td>Renesas Starter Kit for RX62N</td>
</tr>
</tbody>
</table>

(2) RX63N

Table 2.2 Operation Confirmation Conditions

<table>
<thead>
<tr>
<th>Item</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU used for evaluation</td>
<td>RX63N Group (program ROM: 1 MB, RAM: 128 KB)</td>
</tr>
<tr>
<td>Memory used for evaluation</td>
<td>Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I2C Serial EEPROM</td>
</tr>
<tr>
<td>Operating frequency</td>
<td>ICLK: 96 MHz, PCLKB: 48 MHz</td>
</tr>
<tr>
<td>Operating voltage</td>
<td>3.3 V</td>
</tr>
<tr>
<td>Integrated development environment</td>
<td>Renesas Electronics High-performance embedded Workshop Version 4.09.01.007</td>
</tr>
<tr>
<td>C compiler</td>
<td>Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 1.2.1.0)</td>
</tr>
<tr>
<td>Endian mode</td>
<td>Big endian/Little endian</td>
</tr>
<tr>
<td>Sample code version</td>
<td>Ver. 1.13</td>
</tr>
<tr>
<td>Software used</td>
<td>Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01</td>
</tr>
<tr>
<td>Board used</td>
<td>Renesas Starter Kit for RX63N</td>
</tr>
</tbody>
</table>
(3) RX63T

### Table 2.3  Operation Confirmation Conditions

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

(4) RX210

### Table 2.4  Operation Confirmation Conditions

<table>
<thead>
<tr>
<th>Item</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU used for evaluation</td>
<td>RX210 Group (program ROM: 512 KB, RAM: 64 KB)</td>
</tr>
<tr>
<td>Memory used for evaluation</td>
<td>Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I²C Serial EEPROM</td>
</tr>
<tr>
<td>Operating frequency</td>
<td>ICLK: 50 MHz, PCLKB: 25 MHz</td>
</tr>
<tr>
<td>Operating voltage</td>
<td>3.3 V</td>
</tr>
<tr>
<td>Integrated development environment</td>
<td>Renesas Electronics High-performance embedded Workshop Version 4.09.01.007</td>
</tr>
<tr>
<td>C compiler</td>
<td>Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 1.2.1.0)</td>
</tr>
<tr>
<td></td>
<td>Compile option</td>
</tr>
<tr>
<td></td>
<td>Default settings of integrated development environment used as compile options.</td>
</tr>
<tr>
<td></td>
<td>Note: 1. Optimization level: 2; optimization method: optimize for size</td>
</tr>
<tr>
<td>Endian mode</td>
<td>Big endian/Little endian</td>
</tr>
<tr>
<td>Sample code version</td>
<td>Ver. 1.13</td>
</tr>
<tr>
<td>Software used</td>
<td>Renesas Electronics Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01</td>
</tr>
<tr>
<td>Board used</td>
<td>Renesas Starter Kit for RX210</td>
</tr>
</tbody>
</table>
Table 2.5 Operation Confirmation Conditions

<table>
<thead>
<tr>
<th>Item</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU used for evaluation</td>
<td>RX21A Group (program ROM: 512 KB, RAM: 64 KB)</td>
</tr>
<tr>
<td>Memory used for evaluation</td>
<td>Renesas Electronics R1EV24xxx/R1EX24xxx/HN58X24xxx Series I2C Serial EEPROM</td>
</tr>
<tr>
<td>Operating frequency</td>
<td>ICLK: 50 MHz, PCLK: 25 MHz</td>
</tr>
<tr>
<td>Operating voltage</td>
<td>3.3 V</td>
</tr>
<tr>
<td>Integrated development</td>
<td>Renesas Electronics High-performance embedded Workshop Version 4.09.01.007</td>
</tr>
<tr>
<td>environment</td>
<td></td>
</tr>
<tr>
<td>C compiler</td>
<td>Renesas Electronics C/C++ Compiler Package for RX Family (Toolchain 1.2.1.0)</td>
</tr>
<tr>
<td>Compile option</td>
<td>Default settings&lt;sup&gt;1&lt;/sup&gt; of integrated development environment used as compile options.</td>
</tr>
<tr>
<td>Note:</td>
<td>1. Optimization level: 2; optimization method: optimize for size</td>
</tr>
<tr>
<td>Endian mode</td>
<td>Big endian/Little endian</td>
</tr>
<tr>
<td>Sample code version</td>
<td>Ver. 1.13</td>
</tr>
<tr>
<td>Software used</td>
<td>Renesas R1EX24xxx Series Serial EEPROM Control Software (R01AN1075EJ), ver. 1.01</td>
</tr>
<tr>
<td>Board used</td>
<td>HSBRX21AP-B (Hokuto Denshi Co., Ltd.)</td>
</tr>
</tbody>
</table>

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 I2C bus control peripheral functions: the I2C bus interface and serial communication interface simplified I2C bus module.

This application note uses the I2C bus interface.
5. Hardware

5.1 Pins Used

Table 5.1 lists the Pins Used and Their Functions.

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCL (SCL in figure 5.1)</td>
<td>Output</td>
<td>Serial clock output</td>
</tr>
<tr>
<td>SDA (SDA in figure 5.1)</td>
<td>I/O</td>
<td>Serial data I/O</td>
</tr>
</tbody>
</table>

5.2 Reference Circuit

Figure 5.1 shows an example connection between the RX Family I²C bus interface and an I²C 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 I²C Bus Interface and I²C Slave Device
5.3 Controlling Multiple Slave Devices

The sample code supports use of multiple channels. In addition, multiple slave devices with different type names 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.

Example: Devices A and B connected to channel 0 and device C connected to channel 1
Reference: ST: Start condition
SP: Stop condition

Multiple devices on the same channel cannot communicate simultaneously.

Communication on channel 1 possible while channel 0 communication in progress.

Time axis

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

<table>
<thead>
<tr>
<th>Communication Device</th>
<th>Mixed Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fast Mode</td>
</tr>
<tr>
<td>Fast mode</td>
<td>0 to 400 kHz</td>
</tr>
<tr>
<td>Standard mode</td>
<td>0 to 100 kHz</td>
</tr>
</tbody>
</table>
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\textsuperscript{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\textsuperscript{2}C bus single master control.

![Figure 6.1 Software Structure](image)

Legend:
- **API**: User interface function
- **SUB**: Internal function
- **INT**: Interrupt handler
- **SFR**: IP dependent processing
6.2 Operation Overview
This sample code implements I^2C bus single master control using the RX Family MCU I^2C bus interface.

In particular, it implements the following single master protocols.

Table 6.1 Control Protocols

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

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^2C 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.

![Figure 6.2 Master Transmission (Pattern 1) Signals](image-url)
(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.

![Diagram of Pattern 2](image)

**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.

![Diagram of Pattern 3](image)

**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.

![Diagram of Pattern 4](image)

**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.

![Diagram of Master Reception](image)

**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](image)

Legend:
- ST: Start condition generation
- SP: Stop condition generation
- ACK: Acknowledge “0”
- NACK: Acknowledge “1”
- RST: Start condition generation
6.3 Software Operation

Communication is started by calling the start function. After that, \( \text{I}^2\text{C} \) 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 show 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

![Sequence Diagram](image-url)

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, I2C 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 I2C 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 I2C 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>
<thead>
<tr>
<th>No.</th>
<th>Constant Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STS0</td>
<td>R_IIC_STS_NO_INIT</td>
<td>Uninitialized state</td>
</tr>
<tr>
<td>STS1</td>
<td>R_IIC_STS_IDLE</td>
<td>Idle state</td>
</tr>
<tr>
<td>STS2</td>
<td>R_IIC_STS_ST_COND_WAIT</td>
<td>Start condition generation complete wait state</td>
</tr>
<tr>
<td>STS3</td>
<td>R_IIC_STS_SEND_SLVADR_W_WAIT</td>
<td>Slave address [Write] transmission complete wait state</td>
</tr>
<tr>
<td>STS4</td>
<td>R_IIC_STS_SEND_SLVADR_R_WAIT</td>
<td>Slave address [Read] transmission complete wait state</td>
</tr>
<tr>
<td>STS5</td>
<td>R_IIC_STS_SEND_DATA_WAIT</td>
<td>Data transmission complete wait state</td>
</tr>
<tr>
<td>STS6</td>
<td>R_IIC_STS_RECEIVE_DATA_WAIT</td>
<td>Data reception complete wait state</td>
</tr>
<tr>
<td>STS7</td>
<td>R_IIC_STS_SP_COND_WAIT</td>
<td>Stop condition generation complete wait state</td>
</tr>
</tbody>
</table>

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

![State Transition Diagram]

Figure 6.13 Initialization State Transition Diagram

Notation conventions
- Event [condition]/action
  - Events are notated on the left.
  - Actions when events occur are notated on the right.

Notation of state
- [R_IIC_STS_NO_INIT]
  - Uninitialized state
- [R_IIC_STS_IDLE]
  - Idle state

(1) EV0 (call r_iic_drv_init_driver())/
  - Initialization processing
RX600, RX200 Series  I2C Bus Single Master Control Software Using RIIC Serial Interface

Notation conventions

Notation of state

Event [condition]/action
• Events are notated on the left.
• Actions when events occur are
  notated on the right.

[R_IIC_STS_IDLE]
Idle state

(1) EV1 (call _r_iic_drv_generate_start_cond())/
• Start of start condition generation

[R_IIC_STS_ST_COND_WAIT]
Start condition generation complete wait state

(2) EV2 (ICEEI interrupt generation)
[Slave address buffer pointer != NULL]/
• Start of slave address transmission (transfer direction: write)

[R_IIC_STS_SEND_SLVADR_W_WAIT]
Slave address [Write] transmission complete wait state

Pattern 1 operation
(4) EV3 (ICTEI interrupt generation)
[First data buffer pointer != NULL]/
• Start of transmission of the first byte of the 1st data

[R_IIC_STS_SEND_DATA_WAIT]
Data transmission complete wait state

Pattern 2 operation
(5) EV3 (ICTEI interrupt generation)
[First data buffer pointer != NULL &&
second data buffer pointer != NULL]/
• Start of transmission of the first byte of the 2nd data

Pattern 3 operation
(6) EV3 (ICTEI interrupt generation)
[First data buffer pointer != NULL &&
second data buffer pointer != NULL]/
• Start of stop condition generation

[R_IIC_STS_SP_COND_WAIT]
Stop condition generation complete wait state

(7) EV4 (ICTEI interrupt generation)
[First data continuous write in progress]/
• Start of transmission of the second or later byte of the 1st data

Pattern 4 operation
(3) EV2 (ICEEI interrupt generation)
[Slave address buffer pointer == NULL]/
• Start of stop condition generation

(8) EV4 (ICTEI interrupt generation)
[When first data write has completed]/
• Start of transmission of the first byte of the 2nd data

(9) EV4 (ICTEI interrupt generation)
[Second data continuous write in progress]/
• Start of transmission of the second or later byte of the 2nd data

(10) EV4 (ICTEI interrupt generation)
[When second data write has completed]/
• Start of stop condition generation

(11) EV6 (ICEEI interrupt generation)
• Termination processing

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 (gciic_mtx_tbl[])

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

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

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

---

**Event**: An API call or an interrupt  
**Action**: Call r_iic_drv_func()

Acquire current state

Select corresponding function from the protocol state transition table (g_iic_mtx_tbl[])

Clear event flag in g_iic_Event[]

Call selected function

END

Calls the function registered in the protocol state transition table for the current state and the current event.

Returns R_IIC_ERR_OTHER if the corresponding protocol state transition table entry is NULL.

The channel state flag and the device state flag are modified in this processing.

---

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

<table>
<thead>
<tr>
<th>ST</th>
<th>AD6-AD0</th>
<th>/W</th>
<th>/ACK</th>
<th>D7-D0</th>
<th>/ACK</th>
<th>D7-D0</th>
<th>/ACK</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>▲1</td>
<td>▲2</td>
<td>▲3</td>
<td>▲4</td>
<td>▲5</td>
<td></td>
</tr>
</tbody>
</table>

▲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

<table>
<thead>
<tr>
<th>ST</th>
<th>AD6-AD0</th>
<th>/W</th>
<th>/ACK</th>
<th>D7-D0</th>
<th>/ACK</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>▲1</td>
<td>▲2</td>
<td>▲3</td>
<td>▲4</td>
</tr>
</tbody>
</table>

▲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: ICTEI (STOP) interrupt — stop condition detected
(3) **Pattern 3**

<table>
<thead>
<tr>
<th>ST</th>
<th>AD6-AD0</th>
<th>/W</th>
<th>/ACK</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>▲1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▲2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▲3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

▲ 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**

<table>
<thead>
<tr>
<th>ST</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>▲1</td>
<td></td>
</tr>
<tr>
<td>▲2</td>
<td></td>
</tr>
</tbody>
</table>

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

### 6.7.2 Master Reception

<table>
<thead>
<tr>
<th>ST</th>
<th>AD6-AD0</th>
<th>R</th>
<th>/ACK</th>
<th>D7-D0</th>
<th>/ACK</th>
<th>D7-D0</th>
<th>NACK</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>▲1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▲2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▲3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▲4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▲5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

▲ 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

<table>
<thead>
<tr>
<th>ST</th>
<th>AD6-AD0</th>
<th>/W</th>
<th>/ACK</th>
<th>D7-D0</th>
<th>/ACK</th>
<th>RST</th>
<th>AD6-AD0</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>▲1</td>
<td></td>
<td></td>
<td>▲2</td>
<td></td>
<td>▲3</td>
<td>▲4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>/ACK</th>
<th>D7-D0</th>
<th>/ACK</th>
<th>D7-D0</th>
<th>NACK</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>▲5</td>
<td>▲6</td>
<td>▲7</td>
<td>▲8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

▲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 I2C communication information structure.
See section 6.13.1, I2C 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.

![Diagram of data buffer and transmission order](image_url)

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

<table>
<thead>
<tr>
<th>Memory Used</th>
<th>Size</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>4,648 bytes</td>
<td>r_iic_drv_api.c</td>
</tr>
<tr>
<td></td>
<td>(little endian)</td>
<td>r_iic_drv_int.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_iic_drv_sfr.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_iic_drv_sub.c</td>
</tr>
<tr>
<td>RAM</td>
<td>30 bytes</td>
<td>r_iic_drv_api.c</td>
</tr>
<tr>
<td></td>
<td>(little endian)</td>
<td>r_iic_drv_int.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_iic_drv_sfr.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_iic_drv_sub.c</td>
</tr>
<tr>
<td>Maximum usable user stack</td>
<td>84 bytes</td>
<td></td>
</tr>
<tr>
<td>Maximum usable interrupt stack</td>
<td>4 bytes</td>
<td></td>
</tr>
</tbody>
</table>

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

(2) RX210

Table 6.8 Required Memory Sizes

<table>
<thead>
<tr>
<th>Memory Used</th>
<th>Size</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>4,654 bytes</td>
<td>r_iic_drv_api.c</td>
</tr>
<tr>
<td></td>
<td>(little endian)</td>
<td>r_iic_drv_int.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_iic_drv_sfr.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_iic_drv_sub.c</td>
</tr>
<tr>
<td>RAM</td>
<td>30 bytes</td>
<td>r_iic_drv_api.c</td>
</tr>
<tr>
<td></td>
<td>(little endian)</td>
<td>r_iic_drv_int.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_iic_drv_sfr.c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r_iic_drv_sub.c</td>
</tr>
<tr>
<td>Maximum usable user stack</td>
<td>84 bytes</td>
<td></td>
</tr>
<tr>
<td>Maximum usable interrupt stack</td>
<td>4 bytes</td>
<td></td>
</tr>
</tbody>
</table>

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 6.11 File Structure

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\an_r01an1254ej0103_rx_iic</td>
<td>Sample code folder</td>
</tr>
<tr>
<td>\r01an1254ej0103_rx.pdf</td>
<td>Application note</td>
</tr>
<tr>
<td>\source</td>
<td>Folder containing the program</td>
</tr>
<tr>
<td>\r_iic_drv_rx</td>
<td>I²C single master control software folder</td>
</tr>
<tr>
<td>r_iic_drv_api.c</td>
<td>API source file</td>
</tr>
<tr>
<td>r_iic_drv_api.h</td>
<td>API header file</td>
</tr>
<tr>
<td>r_iic_drv_int.c</td>
<td>Interrupt handler source file</td>
</tr>
<tr>
<td>r_iic_drv_int.h</td>
<td>Interrupt handler header file</td>
</tr>
<tr>
<td>r_iic_drv_sfr.h.rx21a</td>
<td>Common register definitions header file (for the RX21A)</td>
</tr>
<tr>
<td>r_iic_drv_sfr.h.rx62n</td>
<td>Common register definitions header file (for the RX62N)</td>
</tr>
<tr>
<td>r_iic_drv_sfr.h.rx63n</td>
<td>Common register definitions header file (for the RX63N)</td>
</tr>
<tr>
<td>r_iic_drv_sfr.h.rx63t</td>
<td>Common register definitions header file (for the RX63T)</td>
</tr>
<tr>
<td>r_iic_drv_sfr.h.rx210</td>
<td>Common register definitions header file (for the RX210)</td>
</tr>
<tr>
<td>r_iic_drv_sfr_rx21a.c</td>
<td>Common register definitions source file (for the RX21A)</td>
</tr>
<tr>
<td>r_iic_drv_sfr_rx62n.c</td>
<td>Common register definitions source file (for the RX62N)</td>
</tr>
<tr>
<td>r_iic_DRV_sfr_rx63n.c</td>
<td>Common register definitions source file (for the RX63N)</td>
</tr>
<tr>
<td>r_iic_DRV_sfr_rx63t.c</td>
<td>Common register definitions source file (for the RX63T)</td>
</tr>
<tr>
<td>r_iic_DRV_sfr_rx210.c</td>
<td>Common register definitions source file (for the RX210)</td>
</tr>
<tr>
<td>r_iic_drv_sub.c</td>
<td>Internal function source file</td>
</tr>
<tr>
<td>r_iic_drv_sub.h</td>
<td>Internal function header file</td>
</tr>
<tr>
<td>\sample</td>
<td>Folder containing the program for verifying EEPROM operation</td>
</tr>
<tr>
<td>sample_background.c</td>
<td>Sample program for verifying EEPROM operation when calling the advance function from the RIIC interrupt handler</td>
</tr>
<tr>
<td>sample_foreground.c</td>
<td>Sample program for verifying EEPROM operation when calling the advance function from the main processing routine</td>
</tr>
</tbody>
</table>

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)

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

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Setting Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_IIC_CH_NUM</td>
<td>(uint8_t)(1)</td>
<td>One plus the maximum number of channels that can be used at the same time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sets to 1 in this sample code.</td>
</tr>
<tr>
<td>REPLY_CNT</td>
<td>(uint32_t)(10000)</td>
<td>Advanced function counter*</td>
</tr>
<tr>
<td>STOP_COND_WAIT</td>
<td>(uint16_t)(1000)</td>
<td>Stop condition generation wait counter*</td>
</tr>
<tr>
<td>BUSCHK_CNT</td>
<td>(uint16_t)(1000)</td>
<td>Bus busy check counter*</td>
</tr>
<tr>
<td>SDACHK_CNT</td>
<td>(uint16_t)(1000)</td>
<td>SDA level check counter*</td>
</tr>
<tr>
<td>GEN_SCLCLK_WAIT</td>
<td>(uint16_t)(1000)</td>
<td>Pseudo clock generation wait counter*</td>
</tr>
<tr>
<td>W_CODE</td>
<td>(uint8_t)(0x00)</td>
<td>Setting value when slave address transfer direction is “write”</td>
</tr>
<tr>
<td>R_CODE</td>
<td>(uint8_t)(0x01)</td>
<td>Setting value when slave address transfer direction is “Read”</td>
</tr>
<tr>
<td>R_IIC_HI</td>
<td>(uint8_t)(0x01)</td>
<td>Port “H”</td>
</tr>
<tr>
<td>R_IIC_LOW</td>
<td>(uint8_t)(0x00)</td>
<td>Port “L”</td>
</tr>
<tr>
<td>R_IIC_OUT</td>
<td>(uint8_t)(0x01)</td>
<td>Port Output</td>
</tr>
<tr>
<td>R_IIC_FALSE</td>
<td>(uint8_t)(0x00)</td>
<td>Flag “OFF”</td>
</tr>
<tr>
<td>R_IIC_TRUE</td>
<td>(uint8_t)(0x01)</td>
<td>Flag “ON”</td>
</tr>
</tbody>
</table>

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.)

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Setting Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_IIC_CHx_LCLK</td>
<td>(uint8_t)(0xED)</td>
<td>I2C bus bit rate low-level register (ICBRL) setting for channel x (x = channel number)*</td>
</tr>
<tr>
<td>R_IIC_CHx_HCLK</td>
<td>(uint8_t)(0xE6)</td>
<td>I2C bus bit rate high-level register (ICBRH) setting for channel x (x = channel number)*</td>
</tr>
<tr>
<td>R_IIC_CHx_ICMR1_INIT</td>
<td>(uint8_t)(0x28)</td>
<td>I2C bus mode register 1 (ICMR1) setting for channel x (x = channel number)*</td>
</tr>
<tr>
<td>R_IIC_IPR_CHx_EEI_INIT</td>
<td>(uint8_t)(0x02)</td>
<td>ICEEIx interrupt priority for channel x (x = channel number)</td>
</tr>
<tr>
<td>R_IIC_IPR_CHx_RXI_INIT</td>
<td>(uint8_t)(0x02)</td>
<td>ICRXIx interrupt priority for channel x (x = channel number)</td>
</tr>
<tr>
<td>R_IIC_IPR_CHx_TXI_INIT</td>
<td>(uint8_t)(0x02)</td>
<td>ICTXIx interrupt priority for channel x (x = channel number)</td>
</tr>
<tr>
<td>R_IIC_IPR_CHx_TEI_INIT</td>
<td>(uint8_t)(0x02)</td>
<td>ICTEIx interrupt priority for channel x (x = channel number)</td>
</tr>
</tbody>
</table>

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 I2C bus mode register 1 (ICMR1)
- I2C bus bit rate low-level register (ICBRL)
- I2C 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 I2C Communication Information Structure

The figure below shows the I2C communication information structure used in the sample code. An instance of this structure must be set up for each slave device used.

```c
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 I2C 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>
<thead>
<tr>
<th>Structure member</th>
<th>Description</th>
</tr>
</thead>
</table>
| *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             | Alignment adjustment members |
| rsv2             | |
| rsv3             | |

Table 6.13 Structure r_iic_drv_info_t 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

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

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

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

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>
<thead>
<tr>
<th>Structure Member</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_IIC_MODE_NONE</td>
<td>No communication state. This mode is transitioned from the uninitialized state or from the idle state.</td>
</tr>
<tr>
<td>R_IIC_MODE_WRITE</td>
<td>Master transmission in progress. This mode is transitioned to by starting communication with the master transmission start function R_IIC_Drv_MasterTx().</td>
</tr>
<tr>
<td>R_IIC_MODE_READ</td>
<td>Master reception in progress. This mode is transitioned to by starting communication with the master reception start function R_IIC_Drv_MasterRx().</td>
</tr>
<tr>
<td>R_IIC_MODE_COMBINED</td>
<td>Master composite operation in progress. This mode is transitioned to by starting communication with the master composite start function R_IIC_Drv_MasterTRx().</td>
</tr>
</tbody>
</table>
### 6.15 Variables

Table 6.20 lists the global variable.

<table>
<thead>
<tr>
<th>Type</th>
<th>Valuable</th>
<th>Description</th>
<th>Function Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t</td>
<td>g_iic_ChStatus[MAX_IIC_CH_NUM]</td>
<td>Channel state flag</td>
<td>R_IIC_Drv_Init</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The communication state defined in table 6.12 can be checked.</td>
<td>R_IIC_Drv_MasterTx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set this variable to R_IIC_NO_INIT at initialization.</td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_GenClk</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_Reset</td>
</tr>
<tr>
<td>r_iic_drv_intenal_event_t</td>
<td>g_iic_Event[MAX_IIC_CH_NUM]</td>
<td>Event flag</td>
<td>R_IIC_Drv_Init</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This flag is set when an interrupt occurs and is cleared by the advance function.</td>
<td>R_IIC_Drv_MasterTx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_A dvance</td>
</tr>
<tr>
<td>r_iic_drv_intenal_info_t</td>
<td>g_iic_InternalInfo[MAX_IIC_CH_NUM]</td>
<td>Internal information management</td>
<td>R_IIC_Drv_Init</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This variable is managed by the sample code and must not be set by the user.</td>
<td>R_IIC_Drv_MasterTx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_A dvance</td>
</tr>
<tr>
<td>uint32_t</td>
<td>g_iic_ReplyCnt[MAX_IIC_CH_NUM]</td>
<td>Advance function counter</td>
<td>R_IIC_Drv_Init</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is the upper limit on the number of calls the advance function.</td>
<td>R_IIC_Drv_MasterTx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It is decremented by the advance function called by the user.</td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It is initialized when an event occurs.</td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If it reaches 0, the channel state flag and the device state flag are set to R_IIC_ERR_NON_REPLY.</td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The counter value can be modified with the REPLY_CNT macro definition.</td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This macro should be set appropriately for the actual user system.</td>
<td>R_IIC_Drv_A dvance</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>g_iic_Api[MAX_IIC_CH_NUM]</td>
<td>API flag</td>
<td>R_IIC_Drv_Init</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This flag is used to prevent multiple calls this sample code’s API.</td>
<td>R_IIC_Drv_MasterTx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It is set when API processing starts and cleared after that processing completes.</td>
<td>R_IIC_Drv_MasterRx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_GenClk</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R_IIC_Drv_Reset</td>
</tr>
</tbody>
</table>
6.16 Functions

Table 6.21 lists the Functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_IIC_Drv_Init()</td>
<td>I2C driver initialization function</td>
</tr>
<tr>
<td>R_IIC_Drv_MasterTx()</td>
<td>Master transmission start function</td>
</tr>
<tr>
<td>R_IIC_Drv_MasterRx()</td>
<td>Master reception start function</td>
</tr>
<tr>
<td>R_IIC_Drv_MasterTRx()</td>
<td>Master composite start function</td>
</tr>
<tr>
<td>R_IIC_Drv_Advance()</td>
<td>Advance function</td>
</tr>
<tr>
<td>R_IIC_Drv_GenClk()</td>
<td>SCL pseudo clock generation function</td>
</tr>
<tr>
<td>R_IIC_Drv_Reset()</td>
<td>I2C driver reset function</td>
</tr>
</tbody>
</table>
6.17 State Transition Diagram

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

![State Transition Diagram]

---

**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>
<thead>
<tr>
<th>State</th>
<th>Channel State Flag</th>
<th>Device State Flag (Communicating Device)</th>
<th>I²C Protocol Operating Mode</th>
<th>Current State of The Protocol Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized state</td>
<td>R_IIC_NO_INIT</td>
<td>R_IIC_NO_INIT</td>
<td>R_IIC_MODE_NONE</td>
<td>R_IIC_STS_NO_INIT</td>
</tr>
<tr>
<td>Idle state</td>
<td>R_IIC_IDLE</td>
<td>R_IIC_IDLE</td>
<td>R_IIC_MODE_NONE</td>
<td>R_IIC_STS_IDLE</td>
</tr>
<tr>
<td>Communication in progress</td>
<td>R_IIC_COMMUNICATION</td>
<td>R_IIC_COMMUNICATION</td>
<td>R_IIC_MODE_WRITE</td>
<td>R_IIC_STS_ST_COND_WAIT</td>
</tr>
<tr>
<td>(master transmission)</td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_SLVADR_W_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_SLVADR_R_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_DATA_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_RECEIVE_DATA_WAIT</td>
</tr>
<tr>
<td>Communication in progress</td>
<td>R_IIC_COMMUNICATION</td>
<td>R_IIC_COMMUNICATION</td>
<td>R_IIC_MODE_READ</td>
<td>R_IIC_STS_ST_COND_WAIT</td>
</tr>
<tr>
<td>(master reception)</td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_SLVADR_W_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_SLVADR_R_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_DATA_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_RECEIVE_DATA_WAIT</td>
</tr>
<tr>
<td>Communication in progress</td>
<td>R_IIC_COMMUNICATION</td>
<td>R_IIC_COMMUNICATION</td>
<td>R_IIC_MODE_COMBINED</td>
<td>R_IIC_STS_ST_COND_WAIT</td>
</tr>
<tr>
<td>(master composite)</td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_SLVADR_W_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_SLVADR_R_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_SEND_DATA_WAIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R_IIC_STS_RECEIVE_DATA_WAIT</td>
</tr>
<tr>
<td>Error state</td>
<td>R_IIC_ERR_PARAM</td>
<td>R_IIC_ERR_PARAM</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>R_IIC_ERR_AL</td>
<td>R_IIC_ERR_AL</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>R_IIC_ERR_NON_REPLY</td>
<td>R_IIC_ERR_NON_REPLY</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>R_IIC_ERR_SCL_GENCLK</td>
<td>R_IIC_ERR_SCL_GENCLK</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>R_IIC_ERR_OTHER</td>
<td>R_IIC_ERR_OTHER</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 6.20 Flag States at State Transitions
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^2C Driver Initialization Function

<table>
<thead>
<tr>
<th><strong>R_IIC_Drv_Init</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Outline</strong></td>
</tr>
<tr>
<td><strong>Header</strong></td>
</tr>
<tr>
<td><strong>Declaration</strong></td>
</tr>
</tbody>
</table>
| **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 (*pRlic_Info.pDevStatus); Sets R_IIC_NO_INIT \*1 |
| **Arguments**       | r_iic_drv_info_t *pRlic_Info ; Pointer to I^2C 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^2C 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.
State flag setting:
\[ R_{\text{IIC}} \_\text{IDLE} \]

Sets the channel state flag and the device state flag.

Pin state: Hi-Z

Assigns pin to port function.\(^1\)

Cancels module stop state of RIIC.

Sets the current state of protocol control to the uninitialized state.

Check channel state flag

Check the channel state flag state.

Channel state

Idle state

Communication in progress

Error state

Uninitialized state:
\[ R_{\text{IIC}} \_\text{NO} \_\text{INIT} \]

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  \textbf{I\textsuperscript{2}C Driver Initialization Function Overview Flowchart (1/2)}
Driver initialization processing

Initializes I2C related registers.

Error?

Successful processing

Set state flag: R_IIC_ERR_OTHER

Error

I2C disable processing

Disable MPC (port function)*1

Internal reset state

Interrupts stopped

• IER: Requests disabled
• IPR: Initialized
• IR: Cleared
• RIIC.ICIER: Disabled

R_IIC_IDLE

R_IIC_ERR_OTHER

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

<table>
<thead>
<tr>
<th>Outline</th>
<th>Master transmission start function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>error_t R_IIC_Drv_MasterTx (r_iic_drv_info_t *pRlic_Info)</td>
</tr>
</tbody>
</table>
| Description    | • Starts master transmission.  
                   • The r_iic_drv_info_t I2C 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 *pRlic_Info ; Pointer to I2C 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 I2C communication information structure is set up correctly. |

*1: Note: The channel state flag and device state flag are not changed.
Remarks
- At the point this function returns, I²C communication has not completed. The advance function must be called to terminate I²C 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 I²C driver setting function. In such cases, call the I²C 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

<table>
<thead>
<tr>
<th>Outline</th>
<th>Master reception start function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h</td>
</tr>
<tr>
<td>Declaration</td>
<td>error_t R_IIC_Drv_MasterRx (r_iic_drv_info_t *pRlic_Info)</td>
</tr>
</tbody>
</table>
| Description        | • Starts master reception.  
                      • The r_iic_drv_info_t I²C 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 *pRlic_Info; Pointer to I²C 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 I²C communication information structure is set up correctly. |
Remarks
- At the point this function returns, I²C communication has not completed. The advance function must be called to terminate I²C 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 I²C driver setting function. In such cases, call the I²C driver initialization function once again before calling the start function.
Error State flag setting:
- R_IIC_COMMUNICATION
- R_IIC_MODE_READ

I²C protocol mode setting:
- R_IIC_MODE_READ

I²C enable processing

State flag setting:
- R_IIC_COMMUNICATION
- R_IIC_NO_INIT
- R_IIC_BUS_BUSY
- R_IIC_ERR_AL
- R_IIC_ERR_NON_REPLY
- R_IIC_ERR_SDA_LOW_HOLD
- R_IIC_ERR_OTHER

Check channel state flag

Channel state

Uninitialized state

Communication in progress

Error state

Idle state

Set up callback function

Initialize RAM

Initialize advance function counter

Check channel state flag

Sets the channel state flag and the device state flag.

If a callback function is defined, specifies that function.

Enables interrupts (RIIC.ICIER: enabled, IR: cleared, IER: requests enabled).
Cancels internal reset, enables MPC (RIIC pin function).\(^1\)

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

Processing to advance communication

Checks the bus busy state.
Generates the start condition.

Error

Successful processing

I²C disable processing

State flag setting:
- R_IIC_COMMUNICATION
- R_IIC_ERR_NON_REPLY

Internal reset state
Interruptions stopped
- IER: Requests disabled
- IPR: Initialized
- IR: Cleared
- RIIC.ICIER: Disabled

Disable MPC (port function)\(^1\)

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 *pRlic_Info) |
| Description | • Starts master composite communication.  
   • The r_iic_drv_info_t I2C 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 *pRlic_Info ; Pointer to I2C 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 I2C communication information structure is set up correctly. |
Remarks

- At the point this function returns, I^2C communication has not completed. The advance function must be called to terminate I^2C 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^2C driver setting function. In such cases, call the I^2C 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 *pRlic_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 *pRlic_Info ; Pointer to I²C 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.

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.
Check channel state flag

Channel state

Uninitialized state

Idle state

Error state

Communication by other device in progress

Communication in progress:

R_IIC_COMMUNICATION

R_IIC_BUS_BUSY

Check events

An event occurred

No event occurred

A

B

Figure 6.28  Advanced Function Overview Flowchart (1/3)
Check advance function counter

Counter == 0 (no response error)

Decrement advance function counter

Counter > 0

Set state flag:
R_IIC_ERR_NON_REPLY

Callback function

R_IIC_COMMUNICATION

If a callback function was set up.

One of the following is executed as communication advance processing. The processing that corresponds to the event that occurred is performed.
- Slave address transmission
- Data transmission
- Data reception
- Restart condition generation
- Stop condition generation
- Communication termination processing

Successful processing, all communication complete

Return value

Successful processing, not all communication complete

R_IIC_COMMUNICATION

Disable MPC (port function)*1
Internal reset state
Interrupts stopped
- IER: Requests disabled
- IPR: Initialized
- IR: Cleared
- RIIC.ICIER: Disabled

I2C disable processing

Set state flag:
R_IIC_FINISH

Callback function

R_IIC_FINISH

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)
NACK detected

Wait for stop condition generation completion

Stop condition detected?

Detected

i²C disable processing
Set state flag: R_IIC_NACK
Set Ret to R_IIC_NACK

Callback function
If a callback function was set up.

Ret

Not detected

i²C disable processing
Set state flag: R_IIC_ERR_NON_REPLY
Set Ret to R_IIC_ERR_NON_REPLY

Other

Error state

Arbitration lost occurred

i²C disable processing
Set state flag: R_IIC_ERR_AL

Callback function
If a callback function was set up.

Ret

Other error

i²C disable processing
Set state flag: R_IIC_ERR_OTHER

Callback function
If a callback function was set up.

Ret

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**

<table>
<thead>
<tr>
<th>Outline</th>
<th>SCL pseudo clock generation function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td><strong>r_iic_drv_api.h, r_iic_drv_sub.h, r_iic_drv_int.h, r_iic_drv_sfr.h</strong></td>
</tr>
<tr>
<td>Declaration</td>
<td>error_t <strong>R_IIC_Drv_GenClk</strong>(r_iic_drv_info_t *pRIic_Info, uint8_t ClkCnt)</td>
</tr>
</tbody>
</table>
| 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 I²C 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  
(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
### I2C Driver Reset Function

#### Outline
I2C 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
```c
definition
```

#### Description
- Resets the I2C 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 I2C 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 I2C communication information structure is set up correctly.

#### Remarks
- To restart communication, it is also necessary to call the I2C 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 I2C bus control register 1 (ICCR1)
   - ST bit in I2C bus control register 2 (ICCR2)
   - BC[2:0] bits in I2C bus mode register 1 (ICMR1)
   - I2C bus status register 1 (ICSR1)
   - I2C bus status register 2 (ICSR2)
   - I2C 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.
Channel state flag setting:
- R_IIC_NO_INIT

I2C disable processing

Internal reset state
- Interrupts stopped
  - IER: Requests disabled
  - IPR: Initialized
  - IR: Cleared
  - RIIC.ICIER: Disabled

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 IC 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.

```c
/* 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.

```c
#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.

```c
#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.

```c
#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 I²C bus mode register 1 (ICMR1)
- I²C bus bit rate low-level register (ICBRL)
- I²C 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.

```c
#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 */

#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 */
```

/* Freq = 400KHz at main system clock = 48MHz */

---

Note1: Division ratio sets it by ICMR1.CKS[2:0].

---
(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.

```c
/* Define channel register. */
#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.PODR.BIT.B3 /* SDA0 Port output data register */
#define R_IIC_PIDR_SCL0 PORT1.PIDR.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.

```c
/* 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.

```c
/* 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.

```c
// 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

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

ICRXI Interrupt Definition

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

ICTEI Interrupt Definition

```c
#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.

---

**Figure 7.1 Example of Recovery Processing Using SCL Pseudo Clock Generation**

- **[1]** Resets the I²C 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.
[1] I2C 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.

   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] I2C driver initialization: R_IIC_Drv_Init()
   If the preceding processing has released SDA and SCL from the low-hold state, call the I2C driver initialization
   function.

   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.

   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 I²C Communication Information Structure

Define the following global I²C 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.

```c
#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 I²C 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 I²C driver initialization function, then call the I²C 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 I²C 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.

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

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.

```c
/* 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>
<thead>
<tr>
<th>MCU</th>
<th>Pin Count</th>
<th>Channel Count</th>
<th>RIIC0</th>
<th>RIIC1</th>
<th>RIIC2</th>
<th>RIIC3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX62N</td>
<td>176, 145</td>
<td>2</td>
<td>P12</td>
<td>P13</td>
<td>P21</td>
<td>P20</td>
</tr>
<tr>
<td></td>
<td>100, 85</td>
<td>1</td>
<td>P12</td>
<td>P13</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>P12</td>
<td>P13</td>
<td>P21</td>
<td>P20</td>
</tr>
<tr>
<td>RX63N/</td>
<td>177, 145</td>
<td>4</td>
<td>P12</td>
<td>P13</td>
<td>P21</td>
<td>P20</td>
</tr>
<tr>
<td>RX631</td>
<td>144</td>
<td>4</td>
<td>P12</td>
<td>P13</td>
<td>P21</td>
<td>P20</td>
</tr>
<tr>
<td></td>
<td>100, 64</td>
<td>2</td>
<td>P12</td>
<td>P13</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RX63T</td>
<td>144</td>
<td>2</td>
<td>PB1</td>
<td>PB2</td>
<td>P25</td>
<td>P26</td>
</tr>
<tr>
<td></td>
<td>120</td>
<td>2</td>
<td>PB1</td>
<td>PB2</td>
<td>P25</td>
<td>P26</td>
</tr>
<tr>
<td></td>
<td>112</td>
<td>1</td>
<td>PB1</td>
<td>PB2</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1</td>
<td>PB1</td>
<td>PB2</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>PB1</td>
<td>PB2</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>1</td>
<td>PB1</td>
<td>PB2</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RX210</td>
<td>145, 100</td>
<td>1</td>
<td>P12, P16</td>
<td>P13, P17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>1</td>
<td>P12, P16</td>
<td>P13, P17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>P16</td>
<td>P17</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>1</td>
<td>P16</td>
<td>P17</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>RX21A</td>
<td>100, 80</td>
<td>2</td>
<td>P12, P16</td>
<td>P13, P17</td>
<td>P21, P20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1</td>
<td>P16</td>
<td>P17</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

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 is 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 I²C 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

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.03</td>
<td>Jan. 30, 2015</td>
<td>First edition issued</td>
</tr>
</tbody>
</table>

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 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 implants 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 against the possibility of physical injury, and injury or damage caused by the 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 require the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for malfunctions or losses occurring as a result of your noncompliance with applicable laws and regulations.

9. Renesas Electronics products and technology may not be used 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 take precautions 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 products” means any product developed or manufactured by or for Renesas Electronics.