

RX Family

RIIC FIT to FSP Migration Guide

Introduction

This application note provides a comprehensive guide on migrating an existing application that uses the I²C Bus Interface (RIIC) Firmware Integration Technology (FIT) module to the RIIC FSP (Flexible Software Package) module.

It includes detailed instructions, configuration steps, and code examples to assist users throughout the migration process.

Target Modules

- FIT Module: `r_riic_rx`
- FSP Module: `r_iic_b_master` and `r_iic_b_slave`

Contents

1. Overview	3
1.1 Stack Diagram	3
1.2 API Functions	4
2. Migrating Project FIT to FSP.....	5
2.1 Creating a new FSP Project	5
2.2 Migrating Configuration	6
2.3 Generate Code	11
2.4 Migrating Application Code	14
2.5 Build and Run the project	19
3. Reference Documents	20
4. Appendices.....	20
4.1 FIT & FSP Environment	20
4.2 Reference FSP Application Code.....	21
4.3 Comparison Reference	32
4.3.1 Compare Functions (Details).....	32
4.3.2 Compare Configuration (Details).....	36
Revision History	37

1. Overview

1.1 Stack Diagram

The figure below illustrates the stack diagram of the RIIC FIT module and the RIIC FSP module.

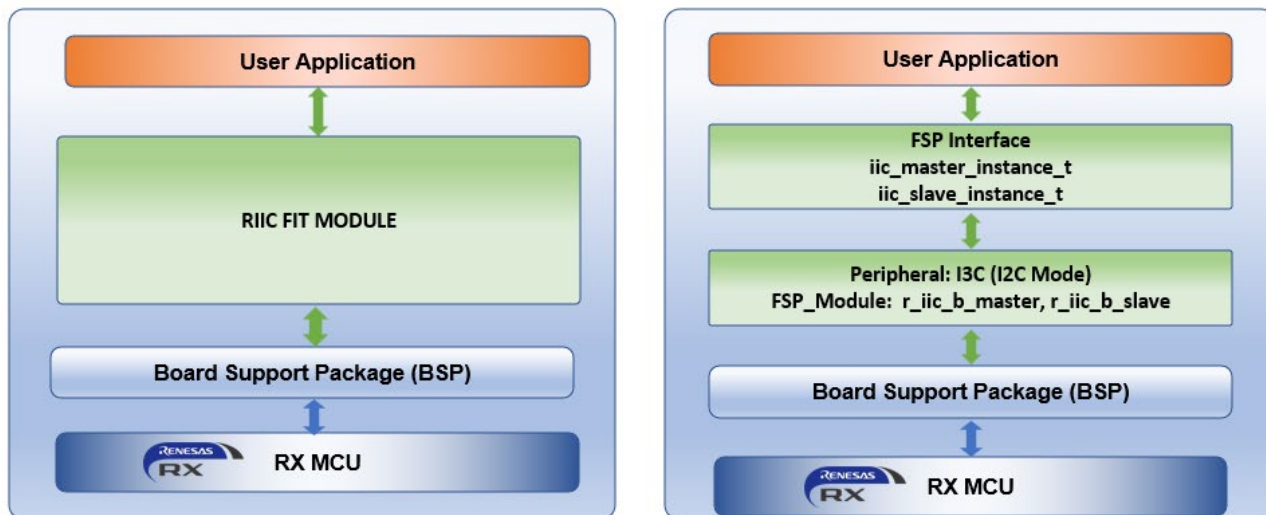


Figure 1-1 RIIC FIT and RIIC FSP stack diagram

In the RIIC FIT implementation, the user application accesses the RIIC peripheral directly through the `r_iic_rx` FIT module on top of the BSP.

In contrast, the RIIC FSP stack introduces an FSP interface layer (`iic_master_instance_t` / `iic_slave_instance_t`) between the application and the hardware, providing better portability and consistency across devices.

1.2 API Functions

This section summarizes the API Functions of the IIC FIT module and the IIC FSP module.

FIT	FSP
R_IIC_Open ⁽¹⁾	open ⁽¹⁾
R_IIC_MasterSend ⁽²⁾	write ⁽²⁾
R_IIC_MasterReceive ⁽²⁾	read ⁽²⁾
R_IIC_SlaveTransfer ⁽³⁾	write ⁽³⁾ /read ⁽³⁾
R_IIC_Close ⁽¹⁾	close ⁽¹⁾
R_IIC_GetStatus ⁽¹⁾	statusGet ⁽²⁾
R_IIC_Control ⁽¹⁾ - Manual START condition - Manual STOP condition - Manual RESTART condition - SDA Hi-Z output - SCL one-shot clock - Module reset	abort ⁽²⁾ Module reset
R_IIC_GetVersion	Not supported in specific modules FSP has common functions to get FSP pack version. R_FSP_VersionGet (in bsp_api.h)
Callback function set in R_IIC_Open	callbackSet ⁽¹⁾
Slave address set in R_IIC_Open	slaveAddressSet ⁽²⁾

Notes:

(1): Using for master and slave.

(2): Using for master only.

(3): Using for slave only.

For more details, please refer to the sections:

[Compare Functions \(Details\)](#) : comparison FIT and FSP.

[Migrate Application Code](#) : example of using FSP IIC API in an application.

2. Migrating Project FIT to FSP

2.1 Creating a new FSP Project

Refer to the reference document “RX FSP Migration Guide” for creating RX FSP project.

To create a new project in e² Studio, navigate to the top bar menu and select:

File > New > Renesas C/C++ Project > Renesas RX.

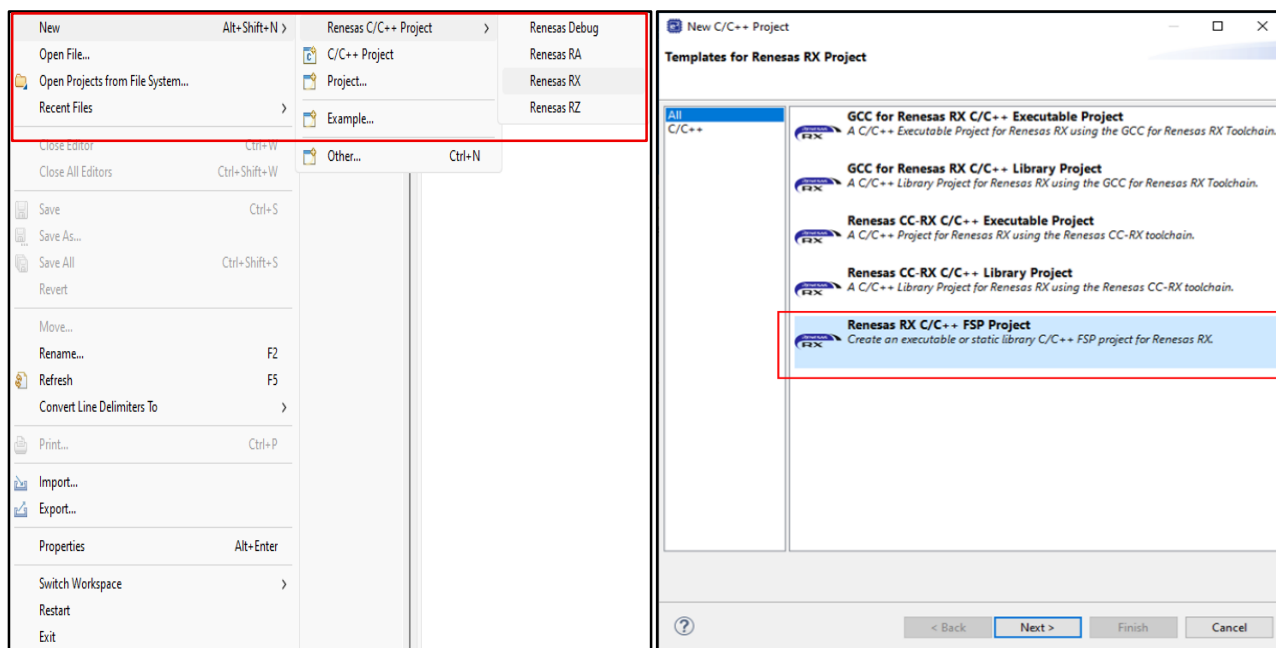


Figure 2-1 Create a new Renesas RX C/C++ FSP Project.

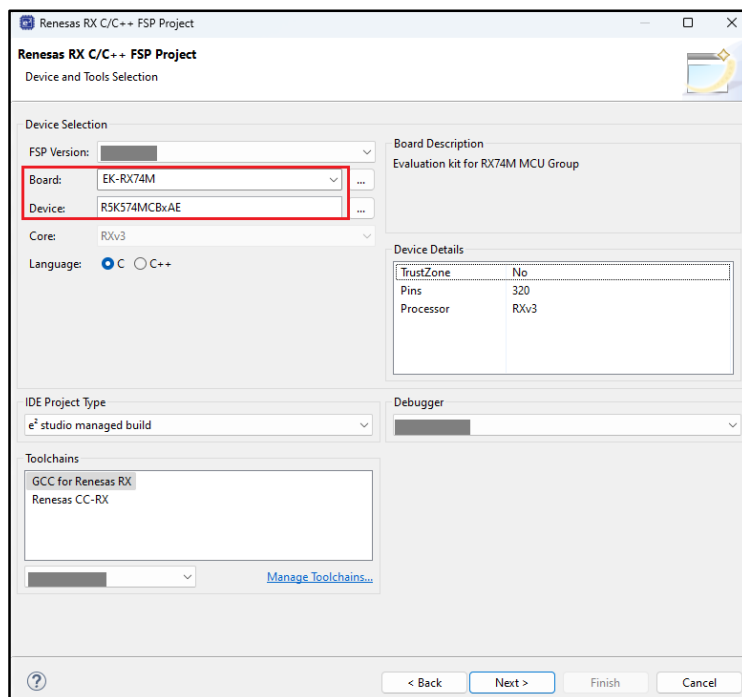


Figure 2-2 Select Board.

2.2 Migrating Configuration

This section describes migration configuration.

Configuration for I2C in this application notes shown as table below:

Configuration	Master	Slave
	Value	Value
Channel	0	1
Rate	Standard	Standard
Slave Address	0x53	0x08
Address Mode	7-bit	7-bit
Callback	g_iic_b_master0_callback	g_iic_b_slave0_callback
SCL Pin	P102	P201
SDA Pin	P103	P200

Steps of updating configuration on FSP Project.

On Configuration Stacks Tab

Click on New Stack -> Connectivity > I2C Master (r_iic_b_master) and I2C Slave (r_iic_b_slave).

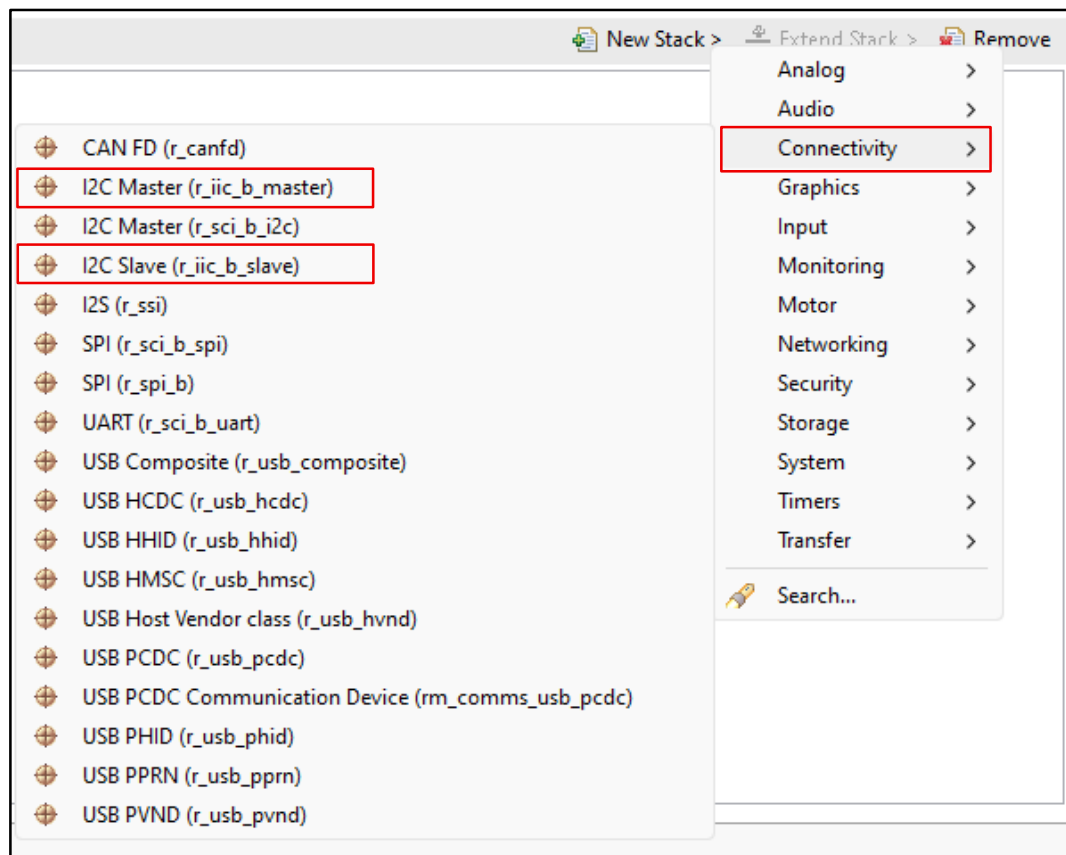


Figure 2-3 Add module I2C Master and I2C Slave in Stack

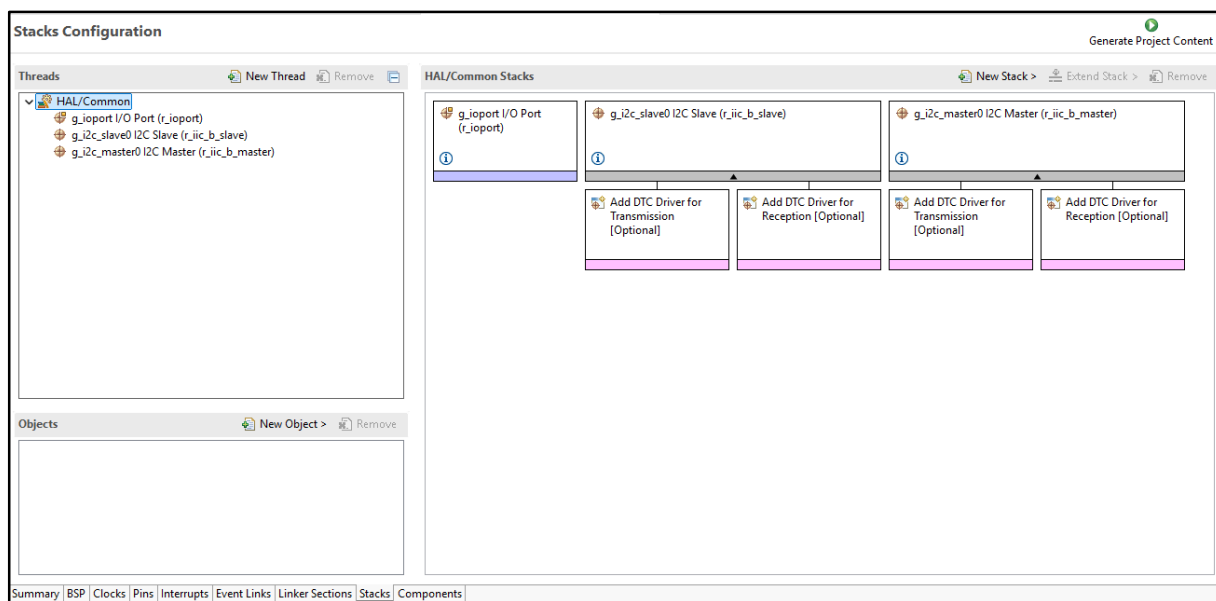


Figure 2-4 I2C Master and I2C Slave in Stack

Click on the "**g_i2c_master0 I2C Master (r_iic_b_master)**" module once it appears.

The "Properties" window will display its properties.

Configure I2C Master according to configuration above.

g_i2c_master0 I2C Master (r_iic_b_master)		
Settings	Property	Value
	▼ Common	
	Parameter Checking	Default (BSP)
	DTC on Transmission and Reception	Disabled
	10-bit slave addressing	Disabled
	▼ Module g_i2c_master0 I2C Master (r_iic_b_master)	
	Name	g_i2c_master0
	Channel	0
	Rate	Standard
	Custom Rate (bps)	0
	Rise Time (ns)	120
	Fall Time (ns)	120
	Duty Cycle (%)	50
	Slave Address	0x00
	Address Mode	7-Bit
	Timeout Mode	Short Mode
	Timeout during SCL Low	Enabled
	Callback	g_iic_b_master0_callback
	Interrupt Priority Level	Priority 2
	▼ Pins	
	I3C_SCL0	P102
	I3C_SDA0	P103

Figure 2-5 Configure I2C Master Properties

Click on the "**g_i2c_slave0 I2C Slave (r_iic_b_slave)**" module once it appears.

The "Properties" window will display its properties.

Configure I2C Slave according to configuration above.

g_i2c_slave0 I2C Slave (r_iic_b_slave)		
Settings	Property	Value
	▼ Common	
	Parameter Checking	Default (BSP)
	DTC on Transmission and Reception	Disabled
	▼ Module g_i2c_slave0 I2C Slave (r_iic_b_slave)	
	Name	g_i2c_slave0
	Channel	1
	Rate	Standard
	Internal Reference Clock	I2C Clock / 1
	Digital Noise Filter Stage Select	3-stage filter
	Slave Address	0x08
	General Call	Disabled
	Address Mode	7-Bit
	Clock Stretching	Disabled
	Callback	g_iic_b_slave0_callback
	▼ Pins	
	SCL1	P201
	SDA1	P200

Figure 2-6 Configure I2C Slave Properties

Requires I3CCLK to be enabled on the Clocks tab.

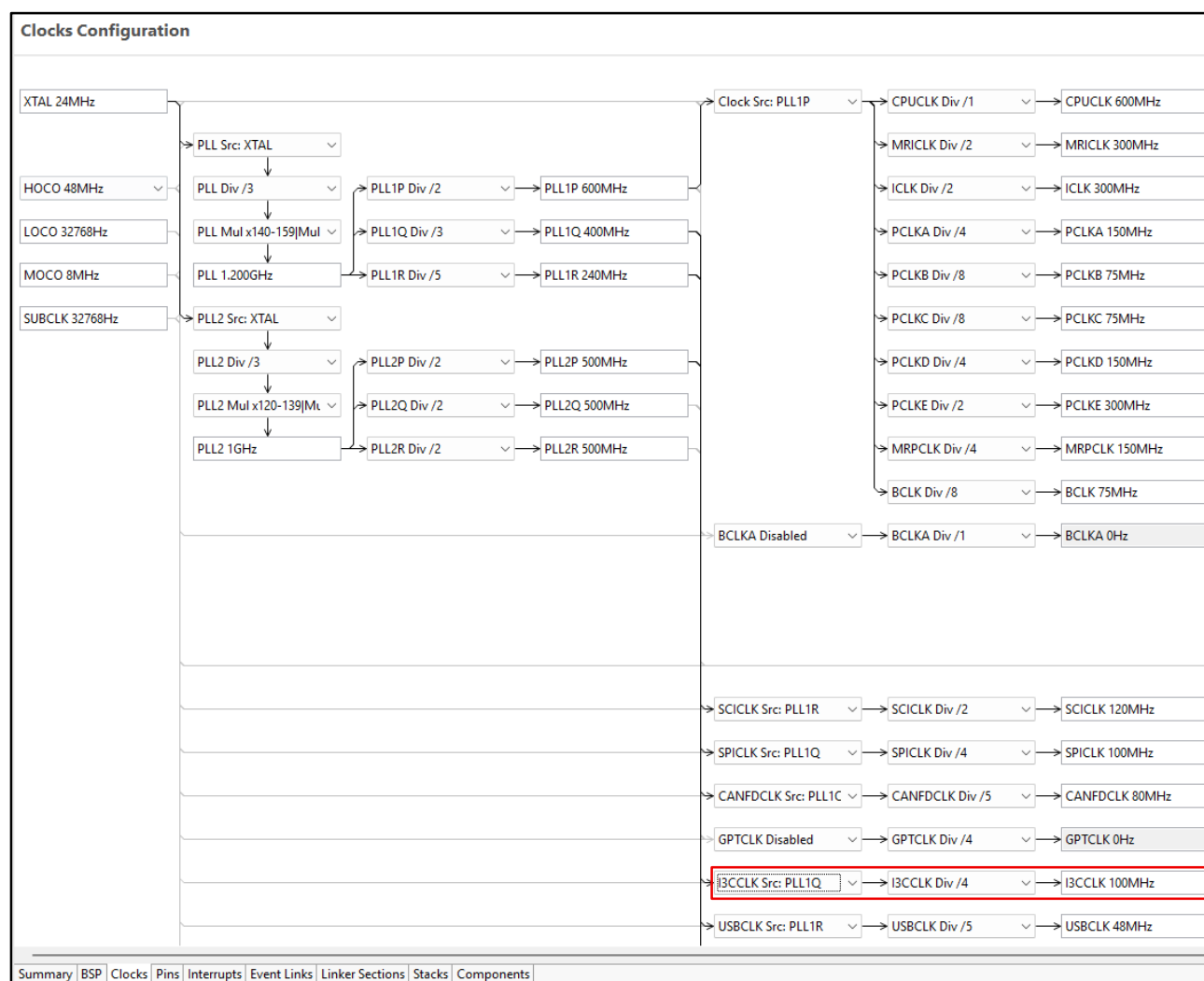


Figure 2-7 Clock Setting

The following sections show the Pin settings configured using the GUI (FSP Configurator window).

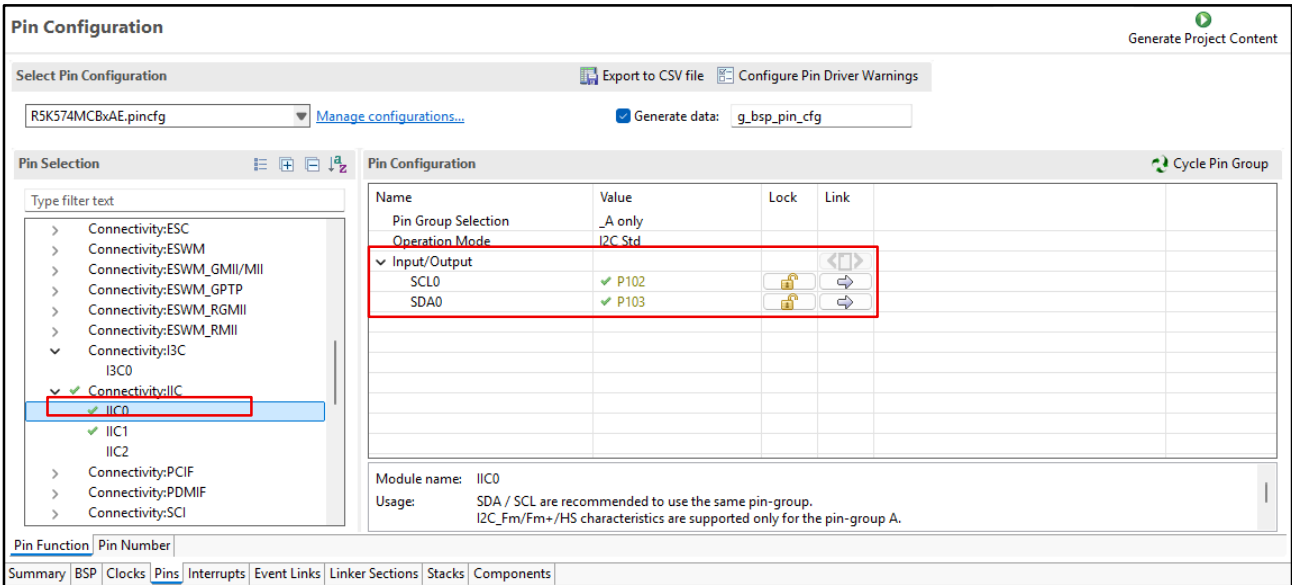


Figure 2-8 IIC0 Pin assignment.

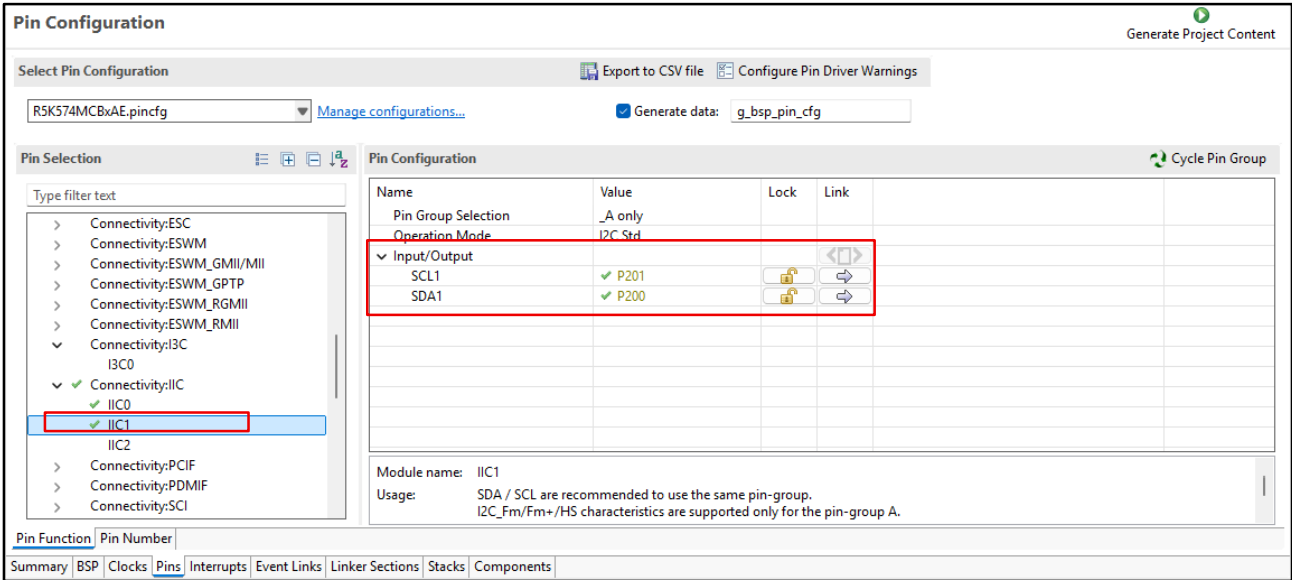


Figure 2-9 IIC1 Pin assignment.

The "Pin Configuration" tab shows the application's pin configuration.

2.3 Generate Code

After completing stacks and HAL configurations and making user selections, click Generate Project Content. This generates the configuration code for selected modules. You can find generated code using the Project Explorer window in e² studio. The configuration.xml file, containing all settings, can be edited in the GUI-based configuration editor for any necessary reconfigurations.

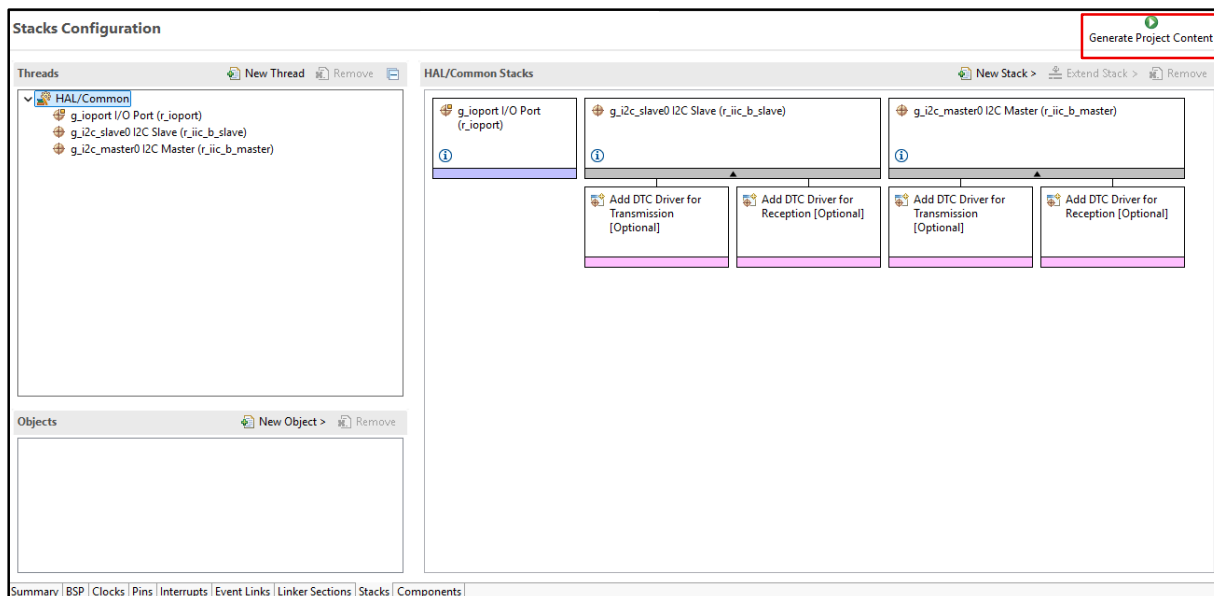
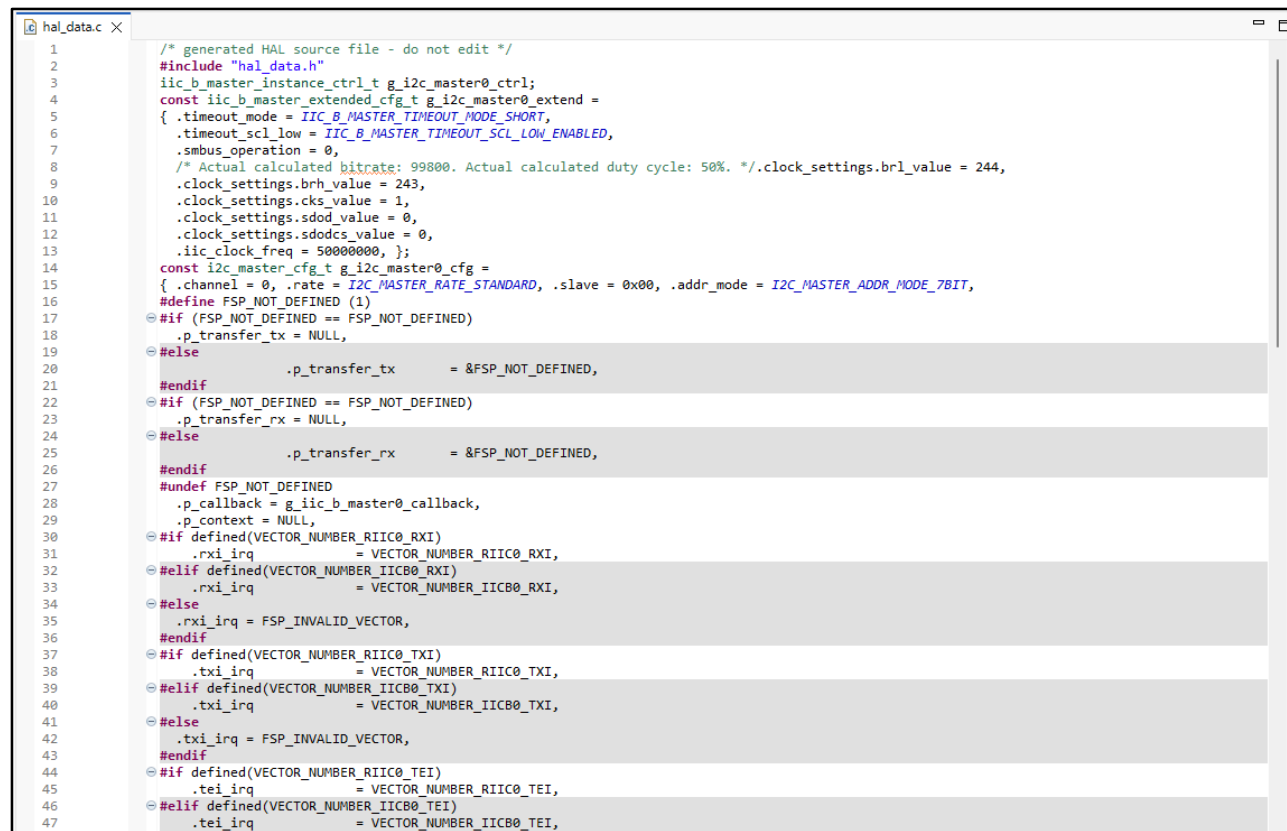


Figure 2-10 Generate Project Content

Once the module has been configured and the files generated, the I2C FSP module is ready to be used in an application.

Generated code in `rx_gen/hal_data.c`.

The following figure shows the generated code for I2C Master according to the configuration in the Properties.



```

1  /* generated HAL source file - do not edit */
2  #include "hal_data.h"
3  iic_b_master_instance_ctrl_t g_i2c_master0_ctrl;
4  const iic_b_master_extended_cfg_t g_i2c_master0_extend =
5  {
6      .timeout_mode = IIC_B_MASTER_TIMEOUT_MODE_SHORT,
7      .timeout_scl_low = IIC_B_MASTER_TIMEOUT_SCL_LOW_ENABLED,
8      .smbus_operation = 0,
9      /* Actual calculated bitrate: 99800. Actual calculated duty cycle: 50%. */ .clock_settings.brh_value = 244,
10     .clock_settings.brh_value = 243,
11     .clock_settings.cks_value = 1,
12     .clock_settings.sdod_value = 0,
13     .clock_settings.sdodcs_value = 0,
14     .iic_clock_freq = 50000000, };
15  const i2c_master_cfg_t g_i2c_master0_cfg =
16  {
17     .channel = 0, .rate = I2C_MASTER_RATE_STANDARD, .slave = 0x00, .addr_mode = I2C_MASTER_ADDR_MODE_7BIT,
18     #define FSP_NOT_DEFINED (1)
19     #if (FSP_NOT_DEFINED == FSP_NOT_DEFINED)
20     .p_transfer_tx = NULL,
21     #else
22     .p_transfer_tx = &FSP_NOT_DEFINED,
23     #endif
24     #if (FSP_NOT_DEFINED == FSP_NOT_DEFINED)
25     .p_transfer_rx = NULL,
26     #else
27     .p_transfer_rx = &FSP_NOT_DEFINED,
28     #endif
29     #undef FSP_NOT_DEFINED
30     .p_callback = g_iic_b_master0_callback,
31     .p_context = NULL,
32     #if defined(VECTOR_NUMBER_RIIC0_RXI)
33     .rx_irq = VECTOR_NUMBER_RIIC0_RXI,
34     #elif defined(VECTOR_NUMBER_IICB0_RXI)
35     .rx_irq = VECTOR_NUMBER_IICB0_RXI,
36     #else
37     .rx_irq = FSP_INVALID_VECTOR,
38     #endif
39     #if defined(VECTOR_NUMBER_RIIC0_TXI)
40     .tx_irq = VECTOR_NUMBER_RIIC0_TXI,
41     #elif defined(VECTOR_NUMBER_IICB0_TXI)
42     .tx_irq = VECTOR_NUMBER_IICB0_TXI,
43     #else
44     .tx_irq = FSP_INVALID_VECTOR,
45     #endif
46     #if defined(VECTOR_NUMBER_RIIC0_TEI)
47     .tei_irq = VECTOR_NUMBER_RIIC0_TEI,
48     #elif defined(VECTOR_NUMBER_IICB0_TEI)
49     .tei_irq = VECTOR_NUMBER_IICB0_TEI,
50     #endif
51 }

```

Figure 2-11 Generated file for I2C Master.

```

hal_data.c X
63 iic_b_slave_instance_ctrl_t g_i2c_slave0_ctrl;
64 const iic_b_slave_extended_cfg_t g_i2c_slave0_extend =
65 {
66     /* Actual delay: 250 ns. */ .clock_settings.brl_value = 25,
67     .clock_settings.digital_filter_stages = 3, .clock_settings.cks_value = 0, };
68 const i2c_slave_cfg_t g_i2c_slave0_cfg =
69 { .channel = 1, .rate = I2C_SLAVE_RATE_STANDARD, .slave = 0x08, .general_call_enable = false, .addr_mode =
70   I2C_SLAVE_ADDR_MODE_7BIT,
71   #if (IIC_B_SLAVE_CFG_DTC_ENABLE)
72   #define FSP_NOT_DEFINED (1)
73   #if (FSP_NOT_DEFINED == FSP_NOT_DEFINED)
74       .p_transfer_tx = NULL,
75   #else
76       .p_transfer_tx = &FSP_NOT_DEFINED,
77   #endif
78   #if (FSP_NOT_DEFINED == FSP_NOT_DEFINED)
79       .p_transfer_rx = NULL,
80   #else
81       .p_transfer_rx = &FSP_NOT_DEFINED,
82   #endif
83   #undef FSP_NOT_DEFINED
84   #endif
85   .p_callback = g_iic_b_slave0_callback,
86   .p_context = NULL,
87   #if defined(VECTOR_NUMBER_RIIC1_RXI)
88       .rx_irq = VECTOR_NUMBER_RIIC1_RXI,
89   #elif defined(VECTOR_NUMBER_IICB1_RXI)
90       .rx_irq = VECTOR_NUMBER_IICB1_RXI,
91   #else
92       .rx_irq = FSP_INVALID_VECTOR,
93   #endif
94   #if defined(VECTOR_NUMBER_RIIC1_TXI)
95       .tx_irq = VECTOR_NUMBER_RIIC1_TXI,
96   #elif defined(VECTOR_NUMBER_IICB1_TXI)
97       .tx_irq = VECTOR_NUMBER_IICB1_TXI,
98   #else
99       .tx_irq = FSP_INVALID_VECTOR,
100   #endif
101   #if defined(VECTOR_NUMBER_I3C1_TEND)
102       .tei_irq = VECTOR_NUMBER_I3C1_TEND,
103   #elif defined(VECTOR_NUMBER_RIIC1_TEI)
104       .tei_irq = VECTOR_NUMBER_RIIC1_TEI,
105   #else
106       .tei_irq = FSP_INVALID_VECTOR,
107   #endif
108   #if defined(VECTOR_NUMBER_I3C1_EEI)
109       .eri_irq = VECTOR_NUMBER_I3C1_EEI,

```

Figure 2-12 Generated file for I2C Slave.

Generated code in file **pin_data.c**

The following figure shows the generated code according to the Pin Configuration.

```

pin_data.c X
/* generated pin source file - do not edit */
#include "bsp_api.h"
#include "r_ioport.h"

const ioport_pin_cfg_t g_bsp_pin_cfg_data[] =
{
    { .pin = BSP_IO_PORT_01_PIN_02, .pin_cfg = ((uint32_t) IOPORT_CFG_DRIVE_1 | (uint32_t) IOPORT_CFG_PERIPHERAL_PIN
      | (uint32_t) IOPORT_PERIPHERAL_I3C) },
    { .pin = BSP_IO_PORT_01_PIN_03, .pin_cfg = ((uint32_t) IOPORT_CFG_DRIVE_1 | (uint32_t) IOPORT_CFG_PERIPHERAL_PIN
      | (uint32_t) IOPORT_PERIPHERAL_I3C) },
    { .pin = BSP_IO_PORT_02_PIN_00, .pin_cfg = ((uint32_t) IOPORT_CFG_DRIVE_1 | (uint32_t) IOPORT_CFG_PERIPHERAL_PIN
      | (uint32_t) IOPORT_PERIPHERAL_I3C) },
    { .pin = BSP_IO_PORT_02_PIN_01, .pin_cfg = ((uint32_t) IOPORT_CFG_DRIVE_1 | (uint32_t) IOPORT_CFG_PERIPHERAL_PIN
      | (uint32_t) IOPORT_PERIPHERAL_I3C) },
    { .pin = BSP_IO_PORT_03_PIN_00, .pin_cfg = ((uint32_t) IOPORT_CFG_DRIVE_1 | (uint32_t) IOPORT_CFG_PERIPHERAL_PIN
      | (uint32_t) IOPORT_PERIPHERAL_DEBUG) },
    { .pin = BSP_IO_PORT_03_PIN_01, .pin_cfg = ((uint32_t) IOPORT_CFG_DRIVE_1 | (uint32_t) IOPORT_CFG_PERIPHERAL_PIN
      | (uint32_t) IOPORT_PERIPHERAL_DEBUG) },
    { .pin = BSP_IO_PORT_03_PIN_02, .pin_cfg = ((uint32_t) IOPORT_CFG_DRIVE_1 | (uint32_t) IOPORT_CFG_PERIPHERAL_PIN
      | (uint32_t) IOPORT_PERIPHERAL_DEBUG) },
    { .pin = BSP_IO_PORT_03_PIN_03, .pin_cfg = ((uint32_t) IOPORT_CFG_DRIVE_1 | (uint32_t) IOPORT_CFG_PERIPHERAL_PIN
      | (uint32_t) IOPORT_PERIPHERAL_DEBUG) }, };

```

Figure 2-13 Generated files for Pin Configuration

2.4 Migrating Application Code

The typical steps for using the RIIC FSP module in this application are as follows:

I2C Master:

1. Configure the I2C Master module using FSP Configuration as described in section 2.2.
2. Initialize the I2C Master by calling the open API.
3. Configure the slave address using slaveAddressSet API.
4. Perform data transmission using the write API.
5. Handle transfer completion and errors via a callback registered with callbackSet.
6. Perform data reception using the read API.
7. Close the I2C Master module using the close API.

```

112  * Test 1: Master (CH0) communication with EEPROM (0x53)
123  static fsp_err_t test_master_ch0_with_eeprom(void)
124  {
125      fsp_err_t ret;
126      uint8_t write_data[7] = {'R', 'e', 'n', 'e', 's', 'a', 's'};
127      uint8_t read_data[8] = {0};
128      uint8_t slave_address = 0x53; // EEPROM address
129
130      printf("\n[TEST] Master (CH0) with EEPROM (0x%02X)\n", slave_address);
131
132      /* Open Master CH0 */
133      ret = g_i2c_master0.p_api->open(g_i2c_master0.p_ctrl, g_i2c_master0.p_cfg);
134      if (FSP_SUCCESS != ret)
135      {
136          printf(" SUCCESS: Master CH0 opened\n");
137      }
138
139      /* Set EEPROM address */
140      ret = g_i2c_master0.p_api->slaveAddressSet(g_i2c_master0.p_ctrl,
141                                              slave_address,
142                                              I2C_MASTER_ADDR_MODE_7BIT);
143      if (FSP_SUCCESS != ret)
144      {
145          /* Bus recovery */
146          ret = g_i2c_master0.p_api->abort(g_i2c_master0.p_ctrl);
147          R_BSP_SoftwareDelay(5, BSP_DELAY_UNITS_MILLISECONDS);
148
149          /* Write data to EEPROM */
150          printf(" Writing data to EEPROM...\n");
151          ret = g_i2c_master0.p_api->write(g_i2c_master0.p_ctrl, write_data, sizeof(write_data), false);
152          if (FSP_SUCCESS != ret)
153          {
154              ret = i2c_wait_transfer(1000);
155              if (FSP_SUCCESS != ret)
156              {
157                  printf(" SUCCESS: Data written to EEPROM\n");
158              }
159
160              /* Wait for EEPROM to write internally */
161              R_BSP_SoftwareDelay(10, BSP_DELAY_UNITS_MILLISECONDS);
162
163              /* Read data back from EEPROM */
164              printf(" Reading data from EEPROM...\n");
165              ret = g_i2c_master0.p_api->read(g_i2c_master0.p_ctrl, read_data, sizeof(read_data), false);
166              if (FSP_SUCCESS != ret)
167              {
168                  ret = i2c_wait_transfer(1000);
169                  if (FSP_SUCCESS != ret)
170                  {
171                      printf(" SUCCESS: Data read from EEPROM\n");
172                      printf(" Data: ");
173                      for (uint32_t i = 0; i < sizeof(read_data); i++)
174                      {
175                          printf("%02X ", read_data[i]);
176                      }
177                      printf("\n");
178
179                      /* Verify data */
180                      if (memcmp(write_data, read_data, sizeof(write_data)) == 0)
181                      {
182                          cleanup:
183                          /* Close Master CH0 */
184                          g_i2c_master0.p_api->close(g_i2c_master0.p_ctrl);
185                          return ret;
186                      }
187                  }
188              }
189          }
190      }
191  }

```

Figure 2-14 Overall FSP I2C Master Application Code

I2C Slave:

1. Configure the I2C Slave module using FSP Configuration.
2. Initialize the I2C Slave by calling the open API.
3. Prepare to receive data by calling the read API.
4. Prepare transmit data by calling the write API.
5. Register a callback function using callbackSet to handle master read/write events.
6. Monitor and process I2C transactions within the callback.
7. Close the I2C Slave module using the close API.

```

227 static fsp_err_t test_slave_ch1_api_comprehensive(void)
228 {
229     fsp_err_t ret;
230     uint8_t slave_rx_buffer[32] = {0};
231     uint8_t slave_tx_buffer[32] = {'S', 'L', 'A', 'V', 'E', '_', 'C', 'H', '1', '_', 'T', 'E', 'S', 'T'};
232
233     print_test_header("SLAVE CH1 API COMPREHENSIVE TEST");
234     printf("Testing Slave API functions with EEPROM on same bus\n");
235     printf("EEPROM Address: 0x53, Slave CH1 Address: 0x%02X\n", g_i2c_slave0.p_cfg->slave);
236
237     /* Test 1: Open Slave CH1 */
238     printf("\n[1/6] Opening Slave CH1...\n");
239     ret = g_i2c_slave0.p_api->open(g_i2c_slave0.p_ctrl, g_i2c_slave0.p_cfg);
240     if (FSP_SUCCESS != ret)
241     {
242         printf(" SUCCESS: Slave CH1 opened\n");
243     }
244
245     /* Test 2: Slave Receive Mode Setup */
246     printf("\n[2/6] Setting up Slave Receive Mode...\n");
247     ret = g_i2c_slave0.p_api->read(g_i2c_slave0.p_ctrl, slave_rx_buffer, sizeof(slave_rx_buffer));
248     if (FSP_SUCCESS != ret)
249     {
250         printf(" SUCCESS: Slave ready to receive data from Master\n");
251     }
252
253     /* Test 3: Slave Transmit Mode Setup */
254     printf("\n[3/6] Setting up Slave Transmit Mode...\n");
255     ret = g_i2c_slave0.p_api->write(g_i2c_slave0.p_ctrl, slave_tx_buffer, sizeof(slave_tx_buffer));
256     if (FSP_SUCCESS != ret)
257     {
258         printf(" SUCCESS: Slave ready to transmit data to Master\n");
259         printf(" TX Buffer: %s\n", slave_tx_buffer);
260     }
261
262     /* Test 5: Bus Monitoring - Wait for activity */
263     printf("\n[5/6] Monitoring I2C Bus Activity (10 seconds)...\n");
264     printf(" Note: EEPROM communication will be visible if Master CH0 is active\n");
265     printf(" Slave CH1 will detect bus events and master requests\n");
266
267     uint32_t monitor_timeout = 10000; // 10 seconds
268     uint32_t event_count = 0;
269
270     while (monitor_timeout > 0)
271     {
272         printf(" Bus monitoring completed. Events detected: %lu\n", event_count);
273     }
274
275     /* Test 6: Slave Close */
276     printf("\n[6/6] Closing Slave CH1...\n");
277     ret = g_i2c_slave0.p_api->close(g_i2c_slave0.p_ctrl);
278     if (FSP_SUCCESS != ret)
279     {
280         printf(" SUCCESS: Slave CH1 closed\n");
281     }
282
283     printf("\nSLAVE API TEST COMPLETED\n");
284     printf("Summary:\n");
285     printf("- Slave Open/Close: Tested\n");
286     printf("- Slave Read Setup: Tested\n");
287     printf("- Slave Write Setup: Tested\n");
288     printf("- Bus Event Monitoring: %lu events\n", event_count);
289     printf("- EEPROM Context: Passive monitoring of bus activity\n");
290
291     return FSP_SUCCESS;
292
293 cleanup:
294     g_i2c_slave0.p_api->close(g_i2c_slave0.p_ctrl);
295     return ret;
296

```

Figure 2-15 Overall FSP I2C Slave Application Code

Open

FIT

FSP

```

uint32_t version_id = 0;
volatile riic_return_t ret;
uint8_t addr_eeprom[1] = {0x53};
uint8_t access_addr[1] = {0x06};
uint8_t mst_send_data[8] = {'R','e','n','e','s','a','s','\0'};
iic_info_m.dev_sts = RIIC_NO_INIT; /*init the iic communication structure*/
iic_info_m.ch_no = 1;
iic_info_m.callbackfunc = &master_call_back;
iic_info_m.cnt2nd = 7;
iic_info_m.cnt1st = 1;
iic_info_m.p_data2nd = mst_send_data;
iic_info_m.p_data1st = access_addr;
iic_info_m.p_slv_addr = addr_eeprom;
printf("iic_info_m.p_data2nd = %s\n",mst_send_data);
ret = R_RIIC_Open(&iic_info_m); /* Open RIIC */
if (RIIC_SUCCESS != ret)
{
    printf("Open I2C failed \n");
    return;
}
else
{
    printf("Open I2C success!!\n");
}

```

```

fsp_err_t ret;
uint8_t write_data[7] = {'R','e','n','e','s','a','s'};
uint8_t read_data[8] = {0};
uint8_t slave_address = 0x53; // EEPROM address

printf("\n[TEST] Master (CH0) with EEPROM (0x%02X)\n", slave_address);

/* Open Master CH0 */
ret = g_i2c_master0.p_api->open(g_i2c_master0.p_ctrl, g_i2c_master0.p_cfg);
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Failed to open Master CH0: 0x%x\n", ret);
    return ret;
}
printf(" SUCCESS: Master CH0 opened\n");

/* Set EEPROM address */
ret = g_i2c_master0.p_api->slaveAddressSet(g_i2c_master0.p_ctrl,
slave_address,
I2C_MASTER_ADDR_MODE_7BIT);
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Failed to set EEPROM address: 0x%x\n", ret);
    goto cleanup;
}

```

Figure 2-16 Master Open Code

```

volatile riic_return_t ret;
uint32_t version_id = 0;
uint8_t slv_send_data[8] = {'R','e','n','e','s','a','s','\0'};
uint8_t slv_store_area[10] = {0x00};

/* Sets IIC Information for Slave Transfer. */
iic_info_s.dev_sts = RIIC_NO_INIT;
iic_info_s.ch_no = 1;
iic_info_s.callbackfunc = &slave_callback;
iic_info_s.cnt2nd = 10;
iic_info_s.cnt1st = 7;
iic_info_s.p_data2nd = slv_store_area;
iic_info_s.p_data1st = slv_send_data;
iic_info_s.p_slv_addr = (uint8_t*)FIT_NO_PTR;
ret = R_RIIC_Open(&iic_info_s);
if (RIIC_SUCCESS != ret)
{
    printf("Open RIIC Failed,ret = %d\n",ret);
    return;
}
else
{
    printf("Open RIIC success\n");
}

```

```

fsp_err_t ret;
uint8_t slave_rx_buffer[32] = {0};
uint8_t slave_tx_buffer[32] = {'S',' ','L',' ','A',' ','V',' ','E',' ','_',' ','C',' ','H',' ','1',' ','_',' ','T',' ','E',' ','S',' ','T'};

print_test_header("SLAVE CH1 API COMPREHENSIVE TEST");
printf("Testing Slave API functions with EEPROM on same bus\n");
printf("EEPROM Address: 0x53, Slave CH1 Address: 0x%02X\n", g_i2c_slave0.p_cfg->slave);

/* Test 1: Open Slave CH1 */
printf("\n[1/6] Opening Slave CH1...\n");
ret = g_i2c_slave0.p_api->open(g_i2c_slave0.p_ctrl, g_i2c_slave0.p_cfg);
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Failed to open Slave CH1: 0x%x\n", ret);
    return ret;
}
printf(" SUCCESS: Slave CH1 opened\n");

```

Figure 2-17 Slave Open Code

Master Write

FIT

FSP

```

ret = R_RIIC_MasterSend(&iic_info_m); /* RIIC Master Send Start */
if (RIIC_SUCCESS != ret)
{
    printf("Master Send failed ,return Value is %d\n",ret);
    R_RIIC_Close(&iic_info_m);
    return;
}

```

```

ret = g_i2c_master0.p_api->write(g_i2c_master0.p_ctrl, write_data, sizeof(write_data), false);
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Write failed: 0x%x\n", ret);
    goto cleanup;
}

```

Figure 2-18 Master writes a message.

Slaves write and read:

FIT

FSP

```

FIT:
ret = R_RIIC_SlaveTransfer(&iic_info_s);
if (RIIC_SUCCESS != ret)
{
    printf("slave transfer failed,ret = %d\n",ret);
    R_RIIC_Close(&iic_info_s);
    return;
}

FSP:
printf("\n[3/6] Setting up Slave Transmit Mode...\n");
ret = g_i2c_slave0.p_api->write(g_i2c_slave0.p_ctrl, slave_tx_buffer, sizeof(slave_tx_buffer));
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Slave write setup failed: 0x%x\n", ret);
    goto cleanup;
}
printf(" SUCCESS: Slave ready to transmit data to Master\n");
printf(" TX Buffer: %s\n", slave_tx_buffer);

printf("\n[2/4] Setting up Slave CH1 to receive...\n");
ret = g_i2c_slave0.p_api->read(g_i2c_slave0.p_ctrl, slave_rx_buffer, sizeof(slave_rx_buffer));
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Slave read setup failed: 0x%x\n", ret);
    goto cleanup;
}

```

Figure 2-19 Slave write and read a message.

Master read

FIT

FSP

```

FIT:
/* RIIC Master Receive Start */
ret = R_RIIC_MasterReceive(&iic_info_m);
if (RIIC_SUCCESS != ret)
{
    printf("Master Receive failed ,return Value is %d\n",ret);
    R_RIIC_Close(&iic_info_m);
    return;
}

FSP:
/* Master reads from Slave */
i2c_transfer_complete = false;
ret = g_i2c_master0.p_api->read(g_i2c_master0.p_ctrl, master_rx_buffer, sizeof(master_rx_buffer), false);
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Master read failed: 0x%x\n", ret);
    goto cleanup;
}

```

Figure 2-20 Master read a message

Master and Slave callback

FIT

FSP

```

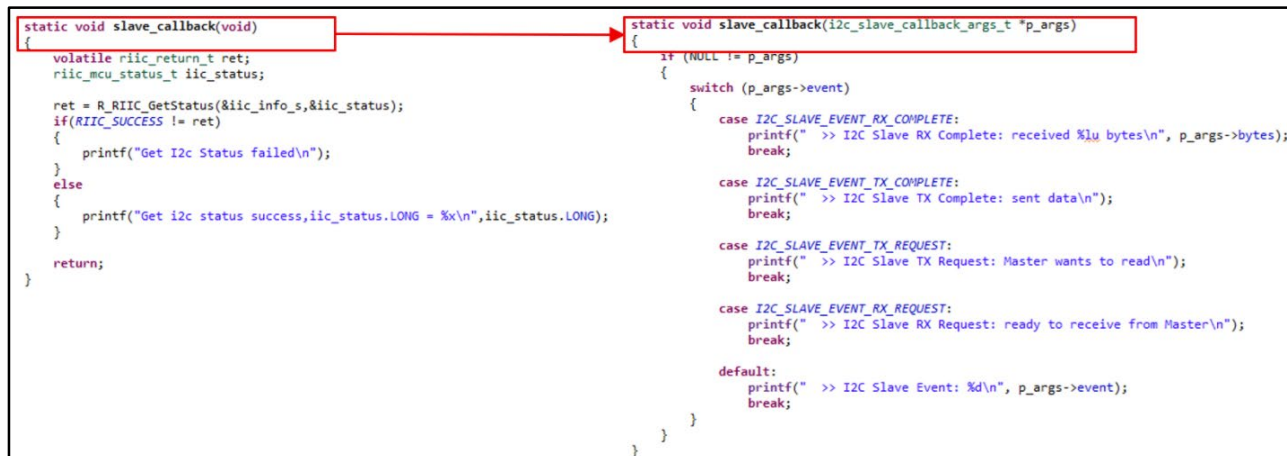
FIT:
static void master_call_back(void)
{
    volatile riic_return_t ret;
    riic_mcu_status_t iic_status;

    /* Get I2C Status */
    ret = R_RIIC_GetStatus(&iic_info_m, &iic_status);
    if (RIIC_SUCCESS != ret)
    {
        printf("Get I2C Status failed\n");
    }
    else
    {
        printf("iic_status.LONG = %d\n",iic_status.LONG);
    }
    return;
}

FSP:
static void master_callback(i2c_master_callback_args_t * p_args)
{
    if (NULL != p_args)
    {
        if (I2C_MASTER_EVENT_TX_COMPLETE == p_args->event)
        {
            i2c_transfer_complete = true;
            i2c_transfer_error = FSP_SUCCESS;
            printf(" >> I2C Master TX Complete (WRITE finished)\n");
        }
        else if (I2C_MASTER_EVENT_RX_COMPLETE == p_args->event)
        {
            i2c_transfer_complete = true;
            i2c_transfer_error = FSP_SUCCESS;
            printf(" >> I2C Master RX Complete (READ finished)\n");
        }
        else if (I2C_MASTER_EVENT_ABORTED == p_args->event)
        {
            i2c_transfer_complete = true;
            i2c_transfer_error = FSP_ERR_ABORTED;
            printf(" >> I2C Master TX Aborted\n");
        }
    }
}

```

Figure 2-21 Master Callback Code



```

static void slave_callback(void)
{
    volatile riic_return_t ret;
    riic_mcu_status_t iic_status;

    ret = R_RIIC_GetStatus(&iic_info_s,&iic_status);
    if(RIIC_SUCCESS != ret)
    {
        printf("Get I2c Status failed\n");
    }
    else
    {
        printf("Get i2c status success,iic_status.LONG = %x\n",iic_status.LONG);
    }
    return;
}

static void slave_callback(i2c_slave_callback_args_t *p_args)
{
    if (NULL != p_args)
    {
        switch (p_args->event)
        {
            case I2C_SLAVE_EVENT_RX_COMPLETE:
                printf(">> I2C Slave RX Complete: received %lu bytes\n", p_args->bytes);
                break;

            case I2C_SLAVE_EVENT_TX_COMPLETE:
                printf(">> I2C Slave TX Complete: sent data\n");
                break;

            case I2C_SLAVE_EVENT_TX_REQUEST:
                printf(">> I2C Slave TX Request: Master wants to read\n");
                break;

            case I2C_SLAVE_EVENT_RX_REQUEST:
                printf(">> I2C Slave RX Request: ready to receive from Master\n");
                break;

            default:
                printf(">> I2C Slave Event: %d\n", p_args->event);
                break;
        }
    }
}

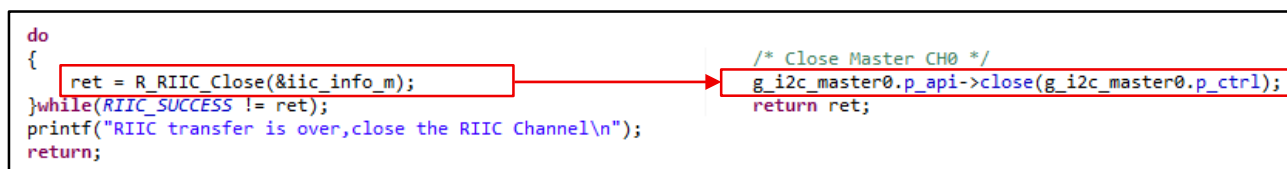
```

Figure 2-22 Slave Callback Code

Close

FIT

FSP



```

do
{
    ret = R_RIIC_Close(&iic_info_m);
}while(RIIC_SUCCESS != ret);
printf("RIIC transfer is over,close the RIIC Channel\n");
return;

/* Close Master CH0 */
g_i2c_master0.p_api->close(g_i2c_master0.p_ctrl);
return ret;

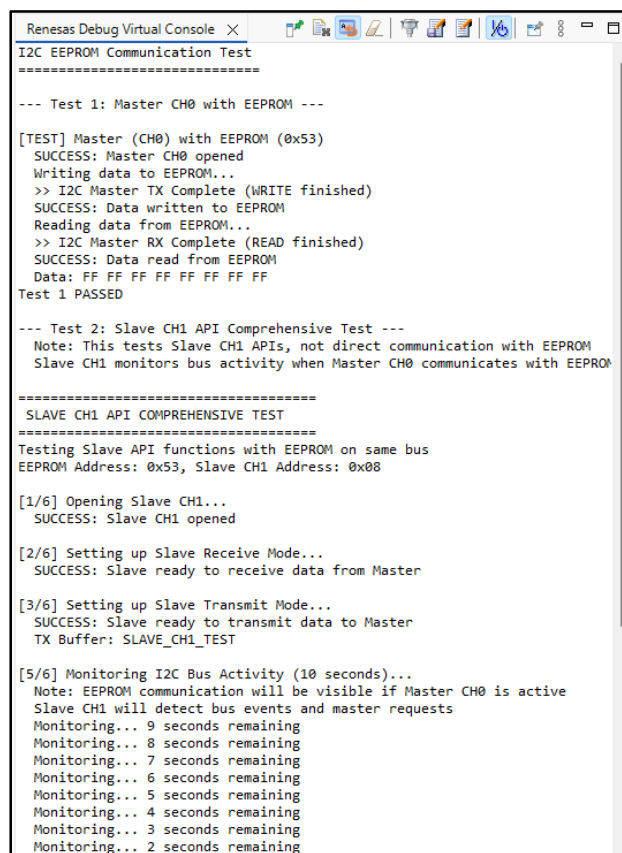
```

Figure 2-23 Close a message.

For more details about FSP sample code, please refer to this section [Reference FSP Application Code](#)

2.5 Build and Run the project

1. Build the sample application, download it to the board, and execute the application using a debugger.
2. After running, the terminal will display logs such as:



```

Renesas Debug Virtual Console
I2C EEPROM Communication Test
=====

--- Test 1: Master CH0 with EEPROM ---

[TEST] Master (CH0) with EEPROM (0x53)
SUCCESS: Master CH0 opened
Writing data to EEPROM...
>> I2C Master TX Complete (WRITE finished)
SUCCESS: Data written to EEPROM
Reading data from EEPROM...
>> I2C Master RX Complete (READ finished)
SUCCESS: Data read from EEPROM
Data: FF FF FF FF FF FF FF FF
Test 1 PASSED

--- Test 2: Slave CH1 API Comprehensive Test ---
Note: This tests Slave CH1 APIs, not direct communication with EEPROM
Slave CH1 monitors bus activity when Master CH0 communicates with EEPROM

=====
SLAVE CH1 API COMPREHENSIVE TEST
=====
Testing Slave API functions with EEPROM on same bus
EEPROM Address: 0x53, Slave CH1 Address: 0x08

[1/6] Opening Slave CH1...
SUCCESS: Slave CH1 opened

[2/6] Setting up Slave Receive Mode...
SUCCESS: Slave ready to receive data from Master

[3/6] Setting up Slave Transmit Mode...
SUCCESS: Slave ready to transmit data to Master
TX Buffer: SLAVE_CH1_TEST

[5/6] Monitoring I2C Bus Activity (10 seconds)...
Note: EEPROM communication will be visible if Master CH0 is active
Slave CH1 will detect bus events and master requests
Monitoring... 9 seconds remaining
Monitoring... 8 seconds remaining
Monitoring... 7 seconds remaining
Monitoring... 6 seconds remaining
Monitoring... 5 seconds remaining
Monitoring... 4 seconds remaining
Monitoring... 3 seconds remaining
Monitoring... 2 seconds remaining

```

Figure 2-24 Output in Renesas Debug Virtual Console.

- **Master (CH0) communication with EEPROM (0x53):**
 1. Open I2C Master CH0 channel
 2. Set slave address (EEPROM)
 3. Write 7 bytes of data: "Renesas"
 4. Wait for write completion → Callback master_callback() signals
 5. Wait 10ms for EEPROM internal write
 6. Read back 8 bytes from EEPROM
 7. Close Master CH0
- **Comprehensive Slave CH1 API Test:**
 1. Open I2C Slave CH1 channel
 2. Set up Slave receive mode (read mode)
 3. Set up Slave transmit mode (write mode) - Buffer: "SLAVE_CH1_TEST"
 4. Monitor I2C bus for 10 seconds:
 - Wait to see if other Master sends data
 - If Slave receives data → Print received data
 5. Close Slave CH1

3. Reference Documents

RX Family RIIC API Using Firmware Integration Technology
Renesas Flexible Software Package (FSP)
RX FSP Migration Guide

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

4. Appendices

4.1 FIT & FSP Environment

This section describes FIT & FSP environment.

Item	Contents
FIT Module	r_riic_rx v3.11
FSP Module	RX FSP v1.0.0 Module: r_riic
Board used	Renesas EK-RX74M (product No.: RTKEK74Mxxxxxxxxx)
Integrated development environment	Renesas Electronics e ² studio
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family GCC for Renesas RX

4.2 Reference FSP Application Code

Add this FSP sample code to file "hal_entry.c"

```
#include "hal_data.h"
/*****
Includes    <System Includes> , "Project Includes"
*****/
#include <stdio.h>
#include <string.h>

/*****
 * Global definition
 *****/

static volatile bool i2c_transfer_complete = false;
static volatile fsp_err_t i2c_transfer_error = FSP_SUCCESS;

/**
 * I2C Master callback function
 */
static void master_callback(i2c_master_callback_args_t * p_args)
{
    if (NULL != p_args)
    {
        if (I2C_MASTER_EVENT_TX_COMPLETE == p_args->event)
        {
            i2c_transfer_complete = true;
            i2c_transfer_error = FSP_SUCCESS;
            printf(" >> I2C Master TX Complete (WRITE finished)\n");
        }
        else if (I2C_MASTER_EVENT_RX_COMPLETE == p_args->event)
        {
            i2c_transfer_complete = true;
            i2c_transfer_error = FSP_SUCCESS;
            printf(" >> I2C Master RX Complete (READ finished)\n");
        }
        else if (I2C_MASTER_EVENT_ABORTED == p_args->event)
        {
            i2c_transfer_complete = true;
            i2c_transfer_error = FSP_ERR_ABORTED;
            printf(" >> I2C Master TX Aborted\n");
        }
    }
}

/**
 * I2C Slave callback function
 */
static void slave_callback(i2c_slave_callback_args_t *p_args)
{
    if (NULL != p_args)
    {
        switch (p_args->event)
        {
            case I2C_SLAVE_EVENT_RX_COMPLETE:
                printf(" >> I2C Slave RX Complete: received %lu bytes\n",
p_args->bytes);

```

```

        break;

    case I2C_SLAVE_EVENT_TX_COMPLETE:
        printf(" >> I2C Slave TX Complete: sent data\n");
        break;

    case I2C_SLAVE_EVENT_TX_REQUEST:
        printf(" >> I2C Slave TX Request: Master wants to read\n");
        break;

    case I2C_SLAVE_EVENT_RX_REQUEST:
        printf(" >> I2C Slave RX Request: ready to receive from
Master\n");
        break;

    default:
        printf(" >> I2C Slave Event: %d\n", p_args->event);
        break;
    }
}

/**
 * Helper function: Print test header
 */
static void print_test_header(const char *title)
{
    printf("\n");
    printf("=====\n");
    printf(" %s\n", title);
    printf("=====\n");
}

/**
 * Helper function: Wait for I2C transfer to complete
 */
static fsp_err_t i2c_wait_transfer(uint32_t timeout_ms)
{
    i2c_transfer_complete = false;
    uint32_t timeout = timeout_ms;

    while (!i2c_transfer_complete && timeout > 0)
    {
        R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_MILLISECONDS);
        timeout--;
    }

    if (!i2c_transfer_complete)
    {
        printf(" ERROR: Transfer timeout!\n");
        return FSP_ERR_TIMEOUT;
    }

    return i2c_transfer_error;
}

/**
 * Test 1: Master (CH0) communication with EEPROM (0x53)
 * Steps:
 * 1. Open I2C Master CH0 channel

```

```

* 2. Set slave address to 0x53 (EEPROM)
* 3. Write 7 bytes of data: "Renesas"
* 4. Wait for write completion → Callback master_callback() signals
* 5. Wait 10ms for EEPROM internal write
* 6. Read back 8 bytes from EEPROM
* 7. Compare data: If match → TEST PASSED
* 8. Close Master CH0
*/
static fsp_err_t test_master_ch0_with_eeprom(void)
{
    fsp_err_t ret;
    uint8_t write_data[7] = {'R', 'e', 'n', 'e', 's', 'a', 's'};
    uint8_t read_data[8] = {0};
    uint8_t slave_address = 0x53; // EEPROM address

    printf("\n[TEST] Master (CH0) with EEPROM (0x%02X)\n", slave_address);

    /* Open Master CH0 */
    ret = g_i2c_master0.p_api->open(g_i2c_master0.p_ctrl, g_i2c_master0.p_cfg);
    if (FSP_SUCCESS != ret)
    {
        printf(" ERROR: Failed to open Master CH0: 0x%x\n", ret);
        return ret;
    }
    printf(" SUCCESS: Master CH0 opened\n");

    /* Set EEPROM address */
    ret = g_i2c_master0.p_api->slaveAddressSet(g_i2c_master0.p_ctrl,
                                                slave_address,
                                                I2C_MASTER_ADDR_MODE_7BIT);
    if (FSP_SUCCESS != ret)
    {
        printf(" ERROR: Failed to set EEPROM address: 0x%x\n", ret);
        goto cleanup;
    }

    /* Bus recovery */
    ret = g_i2c_master0.p_api->abort(g_i2c_master0.p_ctrl);
    R_BSP_SoftwareDelay(5, BSP_DELAY_UNITS_MILLISECONDS);

    /* Write data to EEPROM */
    printf(" Writing data to EEPROM...\n");
    ret = g_i2c_master0.p_api->write(g_i2c_master0.p_ctrl, write_data,
sizeof(write_data), false);
    if (FSP_SUCCESS != ret)
    {
        printf(" ERROR: Write failed: 0x%x\n", ret);
        goto cleanup;
    }

    ret = i2c_wait_transfer(1000);
    if (FSP_SUCCESS != ret)
    {
        printf(" ERROR: Write timeout\n");
        goto cleanup;
    }
    printf(" SUCCESS: Data written to EEPROM\n");

    /* Wait for EEPROM to write internally */
    R_BSP_SoftwareDelay(10, BSP_DELAY_UNITS_MILLISECONDS);

```

```

    /* Read data back from EEPROM */
    printf("    Reading data from EEPROM...\n");
    ret = g_i2c_master0.p_api->read(g_i2c_master0.p_ctrl, read_data,
sizeof(read_data), false);
    if (FSP_SUCCESS != ret)
    {
        printf("    ERROR: Read failed: 0x%x\n", ret);
        goto cleanup;
    }

    ret = i2c_wait_transfer(1000);
    if (FSP_SUCCESS != ret)
    {
        printf("    ERROR: Read timeout\n");
        goto cleanup;
    }

    printf("    SUCCESS: Data read from EEPROM\n");
    printf("    Data: ");
    for (uint32_t i = 0; i < sizeof(read_data); i++)
    {
        printf("%02X ", read_data[i]);
    }
    printf("\n");

    /* Verify data */
    if (memcmp(write_data, read_data, sizeof(write_data)) == 0)
    {
        printf("    SUCCESS: Data verification passed!\n");
    }

cleanup:
    /* Close Master CH0 */
    g_i2c_master0.p_api->close(g_i2c_master0.p_ctrl);
    return ret;
}

/**
 * Test Slave (CH1) with EEPROM (if possible)
 * Note: This may not work if EEPROM is write-only or if bus conflicts
 */
/**
 * Test 2: Comprehensive Slave CH1 API Test
 * Steps:
 * 1. Open I2C Slave CH1 channel
 * 2. Set up Slave receive mode (read mode)
 * 3. Set up Slave transmit mode (write mode) - Buffer: "SLAVE_CH1_TEST"
 * 4. Monitor I2C bus for 10 seconds:
 *    - Wait to see if other Master sends data
 *    - If Slave receives data → Print received data
 * 5. Close Slave CH1
 * Tests all slave API functions and bus monitoring capabilities
 */
static fsp_err_t test_slave_ch1_api_comprehensive(void)
{
    fsp_err_t ret;
    uint8_t slave_rx_buffer[32] = {0};
    uint8_t slave_tx_buffer[32] = {'S', 'L', 'A', 'V', 'E', '_', 'C', 'H', '1',
'_', 'T', 'E', 'S', 'T'};

```

```

print_test_header("SLAVE CH1 API COMPREHENSIVE TEST");
printf("Testing Slave API functions with EEPROM on same bus\n");
printf("EEPROM Address: 0x53, Slave CH1 Address: 0x%02X\n",
g_i2c_slave0.p_cfg->slave);

/* Test 1: Open Slave CH1 */
printf("\n[1/6] Opening Slave CH1...\n");
ret = g_i2c_slave0.p_api->open(g_i2c_slave0.p_ctrl, g_i2c_slave0.p_cfg);
if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Failed to open Slave CH1: 0x%x\n", ret);
    return ret;
}
printf("  SUCCESS: Slave CH1 opened\n");

/* Test 2: Slave Receive Mode Setup */
printf("\n[2/6] Setting up Slave Receive Mode...\n");
ret = g_i2c_slave0.p_api->read(g_i2c_slave0.p_ctrl, slave_rx_buffer,
sizeof(slave_rx_buffer));
if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Slave read setup failed: 0x%x\n", ret);
    goto cleanup;
}
printf("  SUCCESS: Slave ready to receive data from Master\n");

/* Test 3: Slave Transmit Mode Setup */
printf("\n[3/6] Setting up Slave Transmit Mode...\n");
ret = g_i2c_slave0.p_api->write(g_i2c_slave0.p_ctrl, slave_tx_buffer,
sizeof(slave_tx_buffer));
if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Slave write setup failed: 0x%x\n", ret);
    goto cleanup;
}
printf("  SUCCESS: Slave ready to transmit data to Master\n");
printf("  TX Buffer: %s\n", slave_tx_buffer);

/* Test 5: Bus Monitoring - Wait for activity */
printf("\n[5/6] Monitoring I2C Bus Activity (10 seconds)...\n");
printf("  Note: EEPROM communication will be visible if Master CH0 is
active\n");
printf("  Slave CH1 will detect bus events and master requests\n");

uint32_t monitor_timeout = 10000; // 10 seconds
uint32_t event_count = 0;

while (monitor_timeout > 0)
{
    R_BSP_SoftwareDelay(10, BSP_DELAY_UNITS_MILLISECONDS);
    monitor_timeout -= 10;

    /* Check for received data */
    if (slave_rx_buffer[0] != 0)
    {
        printf("  EVENT: Slave received data!\n");
        printf("  Data: ");
        for (uint32_t i = 0; i < sizeof(slave_rx_buffer); i++)
        {

```

```

        if (slave_rx_buffer[i] == 0 && i > 0) break;
        printf("%02X ", slave_rx_buffer[i]);
    }
    printf("\n");
    event_count++;

    /* Clear buffer for next reception */
    memset(slave_rx_buffer, 0, sizeof(slave_rx_buffer));
    ret = g_i2c_slave0.p_api->read(g_i2c_slave0.p_ctrl, slave_rx_buffer,
sizeof(slave_rx_buffer));
    if (FSP_SUCCESS != ret)
    {
        printf("  WARNING: Failed to restart slave receive: 0x%x\n",
ret);
    }

    /* Progress indicator */
    if (monitor_timeout % 1000 == 0)
    {
        printf("  Monitoring... %lu seconds remaining\n", monitor_timeout /
1000);
    }
}

printf("  Bus monitoring completed. Events detected: %lu\n", event_count);

/* Test 6: Slave Close */
printf("\n[6/6] Closing Slave CH1...\n");
ret = g_i2c_slave0.p_api->close(g_i2c_slave0.p_ctrl);
if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Failed to close Slave CH1: 0x%x\n", ret);
    return ret;
}
printf("  SUCCESS: Slave CH1 closed\n");

printf("\nSLAVE API TEST COMPLETED\n");
printf("Summary:\n");
printf("- Slave Open/Close: Tested\n");
printf("- Slave Read Setup: Tested\n");
printf("- Slave Write Setup: Tested\n");
printf("- Bus Event Monitoring: %lu events\n", event_count);
printf("- EEPROM Context: Passive monitoring of bus activity\n");

return FSP_SUCCESS;

cleanup:
    g_i2c_slave0.p_api->close(g_i2c_slave0.p_ctrl);
    return ret;
}

/**
 * Alternative test: Use 2 Masters to verify EEPROM communication
 * Master CH0 writes, Master CH1 reads (if pins allow)
 */
static fsp_err_t test_dual_master_verification(void)
{
    fsp_err_t ret;
    uint8_t write_data[7] = {'T', 'e', 's', 't', 'D', 'a', 't'};

```

```

uint8_t read_data[8] = {0};
uint8_t slave_address = 0x53;

printf("\n[TEST] Dual Master Verification\n");
printf("  Master CH0 writes, Master CH1 reads (if pins allow)\n");

/* Open Master CH0 */
ret = g_i2c_master0.p_api->open(g_i2c_master0.p_ctrl, g_i2c_master0.p_cfg);
if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Failed to open Master CH0: 0x%x\n", ret);
    return ret;
}

/* Open Master CH1 (if available) */
printf("  NOTE: Master CH1 not configured in this setup\n");
printf("  Skipping dual master test (hardware limitation)\n");
g_i2c_master0.p_api->close(g_i2c_master0.p_ctrl);
return FSP_SUCCESS;
}

static fsp_err_t test_slave_master_internal_communication(void)
{
    fsp_err_t ret;
    uint8_t master_tx_data[16] = {'M', 'A', 'S', 'T', 'E', 'R', '_', 'T', 'O',
    '_', 'S', 'L', 'A', 'V', 'E'};
    uint8_t slave_rx_buffer[32] = {0};
    uint8_t slave_tx_data[16] = {'S', 'L', 'A', 'V', 'E', '_', 'T', 'O', '_',
    'M', 'A', 'S', 'T', 'E', 'R'};
    uint8_t master_rx_buffer[32] = {0};

    print_test_header("INTERNAL SLAVE-MASTER COMMUNICATION TEST");
    printf("Testing CH1 Slave with CH0 Master (requires jumper wires)\n");
    printf("Connect: CH0_SDA <-> CH1_SDA, CH0_SCL <-> CH1_SCL\n");

    /* Step 1: Open Slave CH1 first */
    printf("\n[1/4] Opening Slave CH1...\n");
    ret = g_i2c_slave0.p_api->open(g_i2c_slave0.p_ctrl, g_i2c_slave0.p_cfg);
    if (FSP_SUCCESS != ret)
    {
        printf("  ERROR: Failed to open Slave CH1: 0x%x\n", ret);
        printf("  Note: This test requires hardware jumper between CH0 and
CH1\n");
        return ret;
    }
    printf("  SUCCESS: Slave CH1 opened (Address: 0x%02X)\n",
g_i2c_slave0.p_cfg->slave);

    /* Step 2: Setup Slave to receive */
    printf("\n[2/4] Setting up Slave CH1 to receive...\n");
    ret = g_i2c_slave0.p_api->read(g_i2c_slave0.p_ctrl, slave_rx_buffer,
sizeof(slave_rx_buffer));
    if (FSP_SUCCESS != ret)
    {
        printf("  ERROR: Slave read setup failed: 0x%x\n", ret);
        goto cleanup;
    }
    printf("  SUCCESS: Slave ready to receive\n");

    /* Step 3: Open Master CH0 and send data to Slave */
    printf("\n[3/4] Master CH0 sending data to Slave CH1...\n");

```

```

ret = g_i2c_master0.p_api->open(g_i2c_master0.p_ctrl, g_i2c_master0.p_cfg);
if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Failed to open Master CH0: 0x%x\n", ret);
    goto cleanup;
}

/* Set slave address to CH1 slave address */
ret = g_i2c_master0.p_api->slaveAddressSet(g_i2c_master0.p_ctrl,
                                           g_i2c_slave0.p_cfg->slave,
                                           I2C_MASTER_ADDR_MODE_7BIT);

if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Failed to set slave address: 0x%x\n", ret);
    goto cleanup;
}

/* Send data from Master to Slave */
i2c_transfer_complete = false;
ret = g_i2c_master0.p_api->write(g_i2c_master0.p_ctrl, master_tx_data,
sizeof(master_tx_data), false);
if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Master write failed: 0x%x\n", ret);
    goto cleanup;
}

/* Wait for transfer completion */
ret = i2c_wait_transfer(2000);
if (FSP_SUCCESS != ret)
{
    printf("  ERROR: Transfer timeout\n");
    goto cleanup;
}

printf("  SUCCESS: Master sent data to Slave\n");
printf("  Sent: %.s\n", sizeof(master_tx_data), master_tx_data);

/* Wait a bit for slave to process */
R_BSP_SoftwareDelay(100, BSP_DELAY_UNITS_MILLISECONDS);

/* Check if slave received data */
if (slave_rx_buffer[0] != 0)
{
    printf("  SUCCESS: Slave received data!\n");
    printf("  Received: ");
    for (uint32_t i = 0; i < sizeof(slave_rx_buffer); i++)
    {
        if (slave_rx_buffer[i] == 0 && i > 0) break;
        printf("%02X ", slave_rx_buffer[i]);
    }
    printf("\n");

    /* Convert to string for display */
    printf("  As String: %.s\n", sizeof(master_tx_data), slave_rx_buffer);
}
else
{
    printf("  WARNING: Slave did not receive data (check jumper
connections)\n");
}

```

```

}

/* Step 4: Test Slave transmit to Master */
printf("\n[4/4] Testing Slave transmit to Master...\n");

/* Setup Slave to transmit */
memset(slave_rx_buffer, 0, sizeof(slave_rx_buffer)); // Clear RX buffer
ret = g_i2c_slave0.p_api->write(g_i2c_slave0.p_ctrl, slave_tx_data,
sizeof(slave_tx_data));
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Slave write setup failed: 0x%x\n", ret);
    goto cleanup;
}

/* Master reads from Slave */
i2c_transfer_complete = false;
ret = g_i2c_master0.p_api->read(g_i2c_master0.p_ctrl, master_rx_buffer,
sizeof(master_rx_buffer), false);
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Master read failed: 0x%x\n", ret);
    goto cleanup;
}

ret = i2c_wait_transfer(2000);
if (FSP_SUCCESS != ret)
{
    printf(" ERROR: Read transfer timeout\n");
    goto cleanup;
}

printf(" SUCCESS: Master read data from Slave\n");
printf(" Received: ");
for (uint32_t i = 0; i < sizeof(master_rx_buffer); i++)
{
    if (master_rx_buffer[i] == 0 && i > 0) break;
    printf("%02X ", master_rx_buffer[i]);
}
printf("\n");
printf(" As String: %.*s\n", sizeof(slave_tx_data), master_rx_buffer);

/* Verify data */
if (memcmp(slave_tx_data, master_rx_buffer, sizeof(slave_tx_data)) == 0)
{
    printf(" SUCCESS: Data integrity verified!\n");
}
else
{
    printf(" WARNING: Data mismatch detected\n");
}

printf("\nINTERNAL COMMUNICATION TEST COMPLETED\n");
printf("This test validates Slave API functionality with real I2C
traffic\n");

cleanup:
g_i2c_master0.p_api->close(g_i2c_master0.p_ctrl);
g_i2c_slave0.p_api->close(g_i2c_slave0.p_ctrl);
return ret;

```

```

}

/*****
*****//**
 * main() is generated by the RX Configuration editor and is used to generate
threads if an RTOS is used. This function
 * is called by main() when no RTOS is used.

*****
*****/
void hal_entry(void)
{
    fsp_err_t ret;

    /* Initialize FSP modules (pins, ioport, etc.) */
    g_hal_init();

    printf("I2C EEPROM Communication Test\n");
    printf("=====\n");

    /* Test 1: Master CH0 with EEPROM */
    printf("\n--- Test 1: Master CH0 with EEPROM ---\n");
    ret = test_master_ch0_with_eeprom();
    if (FSP_SUCCESS != ret)
    {
        printf("Test 1 FAILED: 0x%x\n", ret);
    }
    else
    {
        printf("Test 1 PASSED\n");
    }

    /* Test 2: Comprehensive Slave CH1 API Test with EEPROM Context */
    printf("\n--- Test 2: Slave CH1 API Comprehensive Test ---\n");
    printf(" Note: This tests Slave CH1 APIs, not direct communication with
EEPROM\n");
    printf(" Slave CH1 monitors bus activity when Master CH0 communicates with
EEPROM\n");
    ret = test_slave_ch1_api_comprehensive();
    if (FSP_SUCCESS != ret)
    {
        printf("Test 2 FAILED: 0x%x\n", ret);
    }
    else
    {
        printf("Test 2 PASSED\n");
    }

    printf("\nAll tests completed.\n");
}

/*****
*****//**
 * This function is called at various points during the startup process. This
implementation uses the event that is
 * called right before main() to set up the pins.
 *
 * @param[in] event Where at in the start up process the code is currently
at

```

```

*****
*****/
void R_BSP_WarmStart(bsp_warm_start_event_t event)
{
    if (BSP_WARM_START_RESET == event)
    {
#if BSP_FEATURE_FLASH_LP_VERSION != 0

        /* Enable reading from data flash. */
        R_FACI_LP->DFLCTL = 1U;

        /* Would normally have to wait tDSTOP(6us) for data flash recovery.
Placing the enable here, before clock and
        * C runtime initialization, should negate the need for a delay since
the initialization will typically take more than 6us. */
#endif
    }

    if (BSP_WARM_START_POST_C == event)
    {
        /* C runtime environment and system clocks are setup. */

        /* Configure pins. */
        R_IOPORT_Open (&IOPORT_CFG_CTRL, &IOPORT_CFG_NAME);
    }
}

/*****
* Function Name : g_iic_b_master0_callback
* Description    : Callback function for I2C Master API
* Arguments      : p_args - pointer to callback arguments
* Return Value   : void
*****/

void g_iic_b_master0_callback(i2c_master_callback_args_t *p_args)
{
    /* Delegate to our master_callback function */
    master_callback(p_args);
}

/*****
End of function g_iic_b_master0_callback
*****/

/*****
* Function Name : g_iic_b_slave0_callback
* Description    : Callback function for I2C Slave API
* Arguments      : p_args - pointer to callback arguments
* Return Value   : void
*****/

void g_iic_b_slave0_callback(i2c_slave_callback_args_t *p_args)
{
    /* Delegate to our slave_callback function */
    slave_callback(p_args);
}

/*****
End of function g_iic_b_slave0_callback
*****/

```

4.3 Comparison Reference

4.3.1 Compare Functions (Details)

Compare API functions (details)

FIT		FSP	
FIT API (Public API)		FSP Interface API (Public API)	FSP Instance API (Private API)
R_RIIC_Open		open	R_IIC_B_MASTER_Open R_IIC_B_SLAVE_Open
R_RIIC_MasterSend		write	R_IIC_B_MASTER_Write
R_RIIC_MasterReceive		read	R_IIC_B_MASTER_Read
R_RIIC_SlaveTransfer		write / read	R_IIC_B_SLAVE_Read R_IIC_B_SLAVE_Write
R_RIIC_Close		close	R_IIC_B_MASTER_Close R_IIC_B_SLAVE_Close
R_RIIC_GetStatus		statusGet	R_IIC_B_MASTER_StatusGet
Callback function set in R_RIIC_Open		callbackSet	R_IIC_B_MASTER_CallbackSet R_IIC_B_SLAVE_CallbackSet
R_RIIC_Control - Manual START condition - Manual STOP condition - Manual RESTART condition - SDA Hi-Z output - SCL one-shot clock - Module reset		abort Module reset	R_IIC_B_MASTER_Abort
Slave address set in R_RIIC_Open		slaveAddressSet	R_IIC_B_MASTER_SlaveAddressSet
R_RIIC_GetVersion		Use R_FSP_VersionGet	-

Compare functions in detail

Master and Slave Open Function	
Description	This function initializes the RIIC FIT module. This function must be called before calling any other API functions.
FIT	R_RIIC_Open (riic_info_t * p_riic_info) FIT uses a common open function.
FSP	Open (i2c_master_ctrl_t * const p_ctrl, i2c_master_cfg_t const * const p_cfg) Open (i2c_slave_ctrl_t * const p_ctrl, i2c_slave_cfg_t const * const p_cfg) FSP separates the Master and Slave APIs.

Master and Slave Callback Set Function	
Description	Callback function.
FIT	R_RIIC_Open (riic_info_t * p_riic_info) In FIT, the callback function is configured in the open function for both master and slave.
FSP	callbackSet (i2c_master_ctrl_t * const p_ctrl, void(*p_callback)(i2c_master_callback_args_t *), void * const p_context, i2c_master_callback_args_t * const p_callback_memory); callbackSet (i2c_slave_ctrl_t * const p_ctrl, void (* p_callback)(i2c_slave_callback_args_t *), void * const p_context, i2c_slave_callback_args_t * const p_callback_memory) In FSP, the callback functions are separate and independently defined for master and slave. The callback function is automatically invoked when the transfer is completed or when an error occurs.

Master and Slave Close Function	
Description	This function completes RIIC communication and releases the RIIC used.
FIT	R_RIIC_Close (riic_info_t * p_riic_info) FIT uses a common close function.
FSP	close (i2c_master_ctrl_t * const p_ctrl) close (i2c_slave_ctrl_t * const p_ctrl) FSP separates the Master and Slave APIs.

Master Receive Function	
Description	Starts master reception. Changes the receive pattern according to the parameters. Operates batched processing until stop condition generation.
FIT	R_RIIC_MasterReceive (riic_info_t * p_riic_info)
FSP	read (i2c_master_ctrl_t * const p_ctrl, uint8_t * const p_dest, uint32_t const bytes, bool const restart) FSP parameter configuration is simpler than FIT.

Master Send Function	
Description	Starts master transmission. Changes the transmit pattern according to the parameters. Operates batched processing until stop condition generation.
FIT	riic_return_t R_RIIC_MasterSend (riic_info_t * p_riic_info)
FSP	write (i2c_master_ctrl_t * const p_ctrl, uint8_t * const p_src, uint32_t const bytes, bool const restart); FSP parameter configuration is simpler than FIT.

Control Function	
Description	This function outputs conditions, Hi-Z from the SDA, and one-shot of the SCL clock. Also it resets the settings of this module. This function is mainly used when a communication error occurs.
FIT	R_RIIC_Control (riic_info_t * p_riic_info, uint8_t ctrl_ptn) FIT provides several low-level control functions, including manual START/STOP/RESTART control, bus hang-up recovery, NACK transmission, SDA Hi-Z output, SCL one-shot clock, and module reset.
FSP	<u>R_IIC_B_MASTER_Abort</u> (i2c_master_ctrl_t * const p_api_ctrl) FSP provides only a reset function in master mode.

Get Status Function	
Description	Returns the state of this module.
FIT	R_RIIC_GetStatus (riic_info_t * p_riic_info, riic_mcu_status_t * p_riic_status)
FSP	<u>statusGet</u> (i2c_master_ctrl_t * const p_ctrl, i2c_master_status_t * p_status) FSP only supports the get-status function for the master.

Transfer Slave Function	
Description	This function performs slave transmission and reception. Changes the transmit and receive pattern according to the parameters.
FIT	R_RIIC_SlaveTransfer (riic_info_t * p_riic_info)
FSP	read (i2c_slave_ctrl_t * const p_ctrl, uint8_t * const p_dest, uint32_t const bytes) write (i2c_slave_ctrl_t * const p_ctrl, uint8_t * const p_src, uint32_t const bytes) FIT uses a single function for slave transmission and reception, with the transfer pattern determined by the parameters, while FSP provides separate read and write APIs.

Additional Functions	
Description	List additional functions or specific function which only have in FIT but don't have in FSP and vice versa.
FIT	Get Version Function: R_RIIC_GetVersion (void) FIT provides a function to return the driver version number at runtime, not present in FSP.
FSP	slaveAddressSet (i2c_master_ctrl_t * const p_ctrl, uint32_t const slave, i2c_master_addr_mode_t const addr_mode) This function is used to set the device address and addressing mode of the slave without reconfiguring the entire bus.

4.3.2 Compare Configuration (Details)

Compare FIT and FSP Configuration:

FIT Configuration	FSP Configuration	
	Master	Slave
Parameter Checking	Common> Parameter Checking	Common> Parameter Checking
Not supported	Common> DTC on Transmission and Reception	Common> DTC on Transmission and Reception
Not supported	10-bit slave addressing	Not available
Not supported	Name	Name
MCU supported channels	Channel	Channel
Setting the default rate for Standard, Fast-mode, and Fast-mode Plus is not supported. The rate is determined by RIIC bsp.	Rate	Rate
RIIC bps(kbps)	Custom Rate (bps)	Not available
SCL rise time	Rise Time (ns)	Not available
SCL fall time	Fall Time (ns)	Not available
Not supported	Duty Cycle (%)	Not available
Slave address 0 for CHx Slave address 1 for CHx Slave address 2 for CHx	Slave Address	Slave Address
Address 0 format for CHx Address 1 format for CHx Address 2 format for CHx	Address Mode	Address Mode
Timeout detection time for CHx	Timeout Mode	Not available
Timeout function for CHx	Timeout during SCL Low	Not available
Not supported	Callback	Callback
CHx RXI INT Priority Level CHx TEI INT Priority Level CHx EEI INT Priority Level CHx TXI INT Priority Level	Interrupt Priority Level	Interrupt Priority Level
Not supported	Not available	Internal Reference Clock
Digital filter for CHx	Not available	Digital Noise Filter Stage
General call address for CHx	Not available	General Call
Not supported	Not available	Clock Stretching
Specifies whether to include processing for port setting in the code.	Not supported	Not supported
Specifies the timeout counter (number of times to perform bus checking)	Not supported	Not supported

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Sep 30, 2026	-	First edition issued

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

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

1. Precaution against Electrostatic Discharge (ESD)

A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity.

Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.

2. Processing at power-on

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

3. Input of signal during power-off state

Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.

4. Handling of unused pins

Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.

5. Clock signals

After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.

6. Voltage application waveform at input pin

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).

7. Prohibition of access to reserved addresses

Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.

8. Differences between products

Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

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

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

Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.

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

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

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

Trademarks

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

Contact information

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