RE01 1500KB, 256KB Group

CMSIS Driver R_I2C Specifications

Summary

This manual describes the detailed specifications of the I2C driver provided in the RE01 1500KB and 256KB Group CMSIS software package (hereinafter called the I2C driver).

Target Device

RE01 1500KB Group

RE01 256KB Group

Contents

1. Overview

This is an I2C driver for RE01 1500KB and 256KB Group devices and is compliant with the ARMS's basic CMSIS software standard. This driver uses the following peripheral functions.

Table 1-1 Peripheral functions used by the R_I2C driver

Peripheral functions	Description
I2C bus interface (RIIC)	Realizes serial communication using the I2C bus (Inter-Integrated Circuit bus) interface method proposed by NXP

2. Driver Configuration

This chapter describes the information required for using this driver.

2.1 File Configuration

This I2C driver conforms to the CMSIS driver package specification and consists of seven files: "Driver_I2C.h" in the ARM CMSIS file storage directory, "r_i2c_cmsis_api.c", "r_i2c_cmsis_api.h", "r_i2c_cfg.h", "R_Drvier_I2C.h", "pin.c" and "pin.h" in the vendor-specific file storage directory. The functions of the files are shown in Table 2-1, and the file configuration is shown in Figure 2-1.

Table 2-1 Roles of the Files of R_I2C Driver

File Name	Description
Driver_I2C.h	This is a CMSIS Driver standard header file.
R_Driver_I2C.h	This is a CMSIS Driver extended header file. To use the I2C driver, it is necessary to include this file.
r_i2c_cmsis_api.c	This is a driver source file. It provides the entities of driver functions. To use the I2C driver, it is necessary to build this file.
r_i2c_cmsis_api.h	This is a driver header file. The macro, type, and prototype declarations to be used in the driver are defined.
r_i2c_cfg.h	This is a configuration definition file. It provides configuration definitions that can be modified by the user.
pin.c	This is a pin setting file. It provides pin assignment processing for various capabilities.
pin.h	This is a pin setting header file.

```
RE01Group CMSIS Package
  ├─CMSIS
  │   └─ Driver
  │        └─ Include
  │             └─ Driver_I2C.h : CMSIS Driver header file
  └─Device
       ├─ CMSIS_Driver
       │    ├─ Src
       │    │    └─r_i2c_cmsis.c : Driver source file
       │    └─ Include
       │         ├─r_i2c_cmsis.h : Driver header file
       │         └─R_Driver_I2C.h : CMSIS Driver extended header file
       ├─ Config
       │    └─ r_i2c_cfg.h : Configuration definition file
       ├─ pin.c : Pin setting file
       └─ pin.h : Pin setting header file
```

Figure 2-1 I2C Driver File Configuration

2.2 Driver APIs

The I2C driver provides channel-specific instances. To use this driver, access APIs by using function pointers to each instance. The list of the I2C driver instances, examples of instance declaration, the APIs contained in the instance, and examples of access to the I2C driver are shown in Table 2-2, Figure 2-2, Table 2-3, and Figure 2-3 to Figure 2-6.

Table 2-2 List of I2C Driver Instances

Instance	Description
ARM_DRIVER_I2C Driver_I2C0	Instance for using RIIC0
ARM_DRIVER_I2C Driver_I2C1	Instance for using RIIC1

```
#include " R_Driver_I2C.h"

// I2C driver instance ( RIIC0 )
extern ARM_DRIVER_I2C Driver_I2C0;
ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
```

Figure 2-2 Example of I2C Driver Instance Declaration

Table 2-3 I2C Driver APIs

API	Description	Reference
Initialize	Initializes the I2C driver (initializes RAM, makes pin settings, and registers interrupts to NVIC).	4.1.1
Uninitialize	Releases the I2C driver (releases the pins). It will also cause a transition to the module stop state if the I2C is not in the state.	4.1.2
PowerControl	Releases the I2C from the module stop state or causes a transition to the mode.	4.1.3
MasterTransmit	Starts master transmission.	4.1.4
MasterReceive	Starts master reception.	4.1.5
SlaveTransmit	Starts slave transmission.	4.1.6
SlaveReceive	Starts slave reception.	4.1.7
GetDataCount	Obtains the transmit data count (slave-mode only).	4.1.8
Control	Executes a control command of the I2C. For the control commands, see "Table 2-4 Command List".	4.1.9
GetStatus	Obtains the status of the I2C.	4.1.10
GetVersion	Obtains the version of the I2C driver.	4.1.11
GetCapabilities	Obtains the capabilities of the I2C driver.	4.1.12

Table 2-4 Command List

Command	Description
ARM_I2C_OWN_ADDRESS	Sets the I2C's own slave address. A 7-bit address should be set as the argument.
ARM_I2C_BUS_SPEED	Sets the I2C bus speed. Either of the following should be set as the argument. ARM_I2C_BUS_SPEED_STANDARD: Standard speed (100 kbps) ARM_I2C_BUS_SPEED_FAST: Fast speed (400 kbps)
ARM_I2C_BUS_CLEAR	Executes bus clear processing.
ARM_I2C_ABORT_TRANSFER	Aborts transmission/reception.

```
#include " R_Driver_I2C.h"

static void callback(uint32_t event);

// I2C driver instance ( RIIC0 )
extern ARM_DRIVER_I2C Driver_I2C0;
ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;

// Transmit data
static uint8_t tx_data[6] = {0x00,0x11,0x12,0x13,0x14,0x15};

main()
{
    (void)i2cDev0->Initialize(callback);                    /* Initializing I2C driver */
    (void)i2cDev0->PowerControl(ARM_POWER_FULL);            /* Release I2C from module stop state */
    (void)i2cDev0->Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_FAST);
                                                            /* Bus speed = Fast speed (400 kbps) */
    (void)i2cDev0->MasterTransmit(0x01, tx_data, 6, false);  /* Starting master transmission
                                                           Transmission destination device address:
                                                           0x01
                                                           Transmission buffer: tx_data
                                                           Transmission size: 6 bytes
                                                           Pending mode: false */
    while(1);
}

/*******************************************************************************************
* callback function
*******************************************************************************************/
static void callback(uint32_t event)
{
    if (ARM_I2C_EVENT_TRANSFER_DONE == event)
    {
        /* Write the processing to be performed when communication completes successfully */
    }
    else
    {
        /* Write the processing to be performed when a communication error occurs */
    }
}
```

Figure 2-3 Example of Access to I2C Driver (Master Transmission)

```
#include " R_Driver_I2C.h"

static void callback(uint32_t event);

// I2C driver instance ( RIIC0 )
extern ARM_DRIVER_I2C Driver_I2C0;
ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;

// Receive Buffer
static uint8_t rx_data[6];

main()
{

    (void)i2cDev0->Initialize(callback);                    /* Initialize the I2C driver */
    (void)i2cDev0->PowerControl(ARM_POWER_FULL);    /* Release the I2C from module stop
state */
    (void)i2cDev0->Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_FAST);
                                                   /* Bus speed = Fast speed (400 kbps) */
    (void)i2cDev0-> MasterReceive (0x01, rx_data, 6, false);  /* Starting master reception
                                                     Reception destination device address:
                                                     0x01
                                                     Reception buffer: rx_data
                                                     Reception size: 6 bytes */
    while(1);
}

/******************************************************************************************
* callback function
******************************************************************************************/
static void callback(uint32_t event)
{
    if (ARM_I2C_EVENT_TRANSFER_DONE == event)
    {
        /* Write the processing to be performed when communication completes successfully */
    }
    else
    {
        /* Write the processing to be performed when a communication error occurs */
    }
}
```

Figure 2-4 Example of Access to I2C Driver (Master Reception)

```
#include " R_Driver_I2C.h"

static void callback(uint32_t event);

// I2C driver instance ( RIIC0 )
extern ARM_DRIVER_I2C Driver_I2C0;
ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;

// Transmit data
static uint8_t tx_data[6] = {0x00,0x11,0x12,0x13,0x14,0x15};

main()
{
    uint32_t bus_speed = ARM_I2C_BUS_SPEED_FAST;

    (void)i2cDev0->Initialize(callback);                   /* Initializes I2C driver */
    (void)i2cDev0->PowerControl(ARM_POWER_FULL);    /* Releases the I2C from module stop
state */
    (void)i2cDev0-> SlaveTramsmit (tx_data, 6);           /* Start slave transmission
(transmission size: 6 bytes) */
    while(1);
}

/******************************************************************************************
* callback function
******************************************************************************************/
static void callback(uint32_t event)
{
    switch (event)
    {
    case ARM_I2C_EVENT_SLAVE_RECEIVE:
        /* Write the processing to be performed upon receiving address + W */
    break;

    case ARM_I2C_EVENT_SLAVE_TRANSMIT
        /* Write the processing to be performed upon receiving address + R */
    break;

    case ARM_I2C_EVENT_TRANSFER_DONE | ARM_I2C_EVENT_GENERAL_CALL:
    case ARM_I2C_EVENT_TRANSFER_DONE:
        /* Write the processing to be performed when communication completes successfully */
    break;

    default:
        /* Write the processing to be performed when a communication error occurs */
    break;
    }
}
```

Figure 2-5 Example of Access to I2C Driver (Slave Reception)

```
#include " R_Driver_I2C.h"

static void callback(uint32_t event);

// I2C driver instance ( RIIC0 )
extern ARM_DRIVER_I2C Driver_I2C0;
ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;

// Receive Buffer
static uint8_t rx_data[6];

main()
{
    uint32_t bus_speed = ARM_I2C_BUS_SPEED_FAST;

    (void)i2cDev0->Initialize(callback);                    /* Initialize I2C driver */
    (void)i2cDev0->PowerControl(ARM_POWER_FULL);    /* Release the I2C from module stop
state */
    (void)i2cDev0-> SlaveReceive (rx_data, 6);            /* Start slave reception
(reception size: 6 bytes) */
    while(1);
}

/******************************************************************************************
* callback function
******************************************************************************************/
static void callback(uint32_t event)
{
    switch (event)
    {
    case ARM_I2C_EVENT_SLAVE_RECEIVE:
        /* Write the processing to be performed upon receiving address + W */
    break;

    case ARM_I2C_EVENT_SLAVE_TRANSMIT
        /* Write the processing to be performed upon receiving address + R */
    break;

    case ARM_I2C_EVENT_TRANSFER_DONE | ARM_I2C_EVENT_GENERAL_CALL:
    case ARM_I2C_EVENT_TRANSFER_DONE:
        /* Write the processing to be performed when communication completes successfully */
    break;

    default:
        /* Write the processing to be performed when a communication error occurs */
    break;
    }
}
```

Figure 2-6   Example of Access to I2C Driver (Slave Reception)

2.3 Pin Configuration

The pins to be used by this driver are set and released with the R_RIIC_Pinset_CHn (n = 0, 1) and R_RIIC_Pinclr_CHn functions in pin.c. The R_RIIC_Pinset_CHn and R_RIIC_Pinclr_CHn functions are called from the Initialize and Uninitialize functions.

Select the pin to be used by editing the R_RIIC_Pinset and R_RIIC_Pinclr functions of pin.c. Figure 2-7 shows examples of pin settings.

```
/******************************************************************************//**
* @brief This function sets Pin of RIIC0.
*******************************************************************************/
/* Function Name : R_RIIC_Pinset_CH0 */
void R_RIIC_Pinset_CH0(void)  // @suppress("Source file naming") @suppress("API function naming")
{
    /* Disable protection for PFS function (Set to PWPR register) */
    R_SYS_RegisterProtectDisable(SYSTEM_REG_PROTECT_MPC);

    /* Set P810 as SCL0 */
    /* SCL0 : P810 */
    PFS->P810PFS_b.ASEL = 0U;  /* 0: Do not use as an analog pin, 1: Use as an analog pin. */
    PFS->P810PFS_b.ISEL = 0U;/* 0: Do not use as an IRQn input pin,  1: Use as an IRQn input pin. */
    PFS->P810PFS_b.DSCR = 0x2U;  /* When using RIIC : DSCR = 10b */
    PFS->P810PFS_b.PSEL = R_PIN_PRV_RIIC_PSEL;
    PFS->P810PFS_b.PMR  = 1U;/* 0: Use the pin as a general I/O port,
                                1: Use the pin as a peripheral module. */

    /* Set P809 as SDA0 */
    /* SDA0 : P809 */
    PFS->P809PFS_b.ASEL = 0U;  /* 0: Do not use as an analog pin, 1: Use as an analog pin. */
    PFS->P809PFS_b.ISEL = 0U;  /* 0: Do not use as an IRQn input pin,
                                  1: Use as an IRQn input pin. */
    PFS->P809PFS_b.DSCR = 0x2U;  /* When using RIIC : DSCR = 10b */
    PFS->P809PFS_b.PSEL = R_PIN_PRV_RIIC_PSEL;
    PFS->P809PFS_b.PMR  = 1U;  /* 0: Use the pin as a general I/O port,
                                  1: Use the pin as a peripheral module. */

    /* Enable protection for PFS function (Set to PWPR register) */
    R_SYS_RegisterProtectEnable(SYSTEM_REG_PROTECT_MPC);

}/* End of function R_RIIC_Pinset_CH0() */

/******************************************************************************//**
* @brief This function clears the pin setting of RIIC0.
*******************************************************************************/
/* Function Name : R_RIIC_Pinclr_CH0 */
void R_RIIC_Pinclr_CH0(void)  // @suppress("Source file naming") @suppress("API function naming")
{
    /* Disable protection for PFS function (Set to PWPR register) */
    R_SYS_RegisterProtectDisable(SYSTEM_REG_PROTECT_MPC);

    /* Release SCL0 pin */
    /* SCL0 : P810 */
    PFS->P810PFS &= R_PIN_PRV_CLR_MASK;

    /* Release SDA0 pin */
    /* SDA0 : P809 */
    PFS->P809PFS &= R_PIN_PRV_CLR_MASK;

    /* Enable protection for PFS function (Set to PWPR register) */
    R_SYS_RegisterProtectEnable(SYSTEM_REG_PROTECT_MPC);
}/* End of function R_RIIC_Pinclr_CH0() */
```

Figure 2-7 Examples of Setting Pin

2.4 Registering I2C Interrupts to NVIC

It is necessary to register the interrupts used for communication control to the nested vectored interrupt controller (hereinafter referred to as NVIC) in r_system_cfg.h. For details, refer to "Interrupt Control" in the RE01 1500KB, 256KB Group Getting Started Guide to Development Using CMSIS Package.

Table 2-5 shows the definition of NVIC registration for each intended use and Figure 2-8 shows the coding example for registering the interrupts to the NVIC.

Table 2-5 Definitions of NVIC Registration

Interrupts	Definition of NVIC Registration(n=0,1)
Receive data full interrupt (RXI)	SYSTEM_CFG_EVENT_NUMBER_IICn_RXI
Transmit end interrupt (TEI)	SYSTEM_CFG_EVENT_NUMBER_IICn_TEI
Transmit data empty interrupt (TXI)	SYSTEM_CFG_EVENT_NUMBER_IICn_TXI
Communication error/event generation interrupt (EEI)	SYSTEM_CFG_EVENT_NUMBER_IICn_EEI

```
• • •
#define SYSTEM_CFG_EVENT_NUMBER_ADC140_WCMPM
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 0/4/8/12/16/20/24/28 only */
#define SYSTEM_CFG_EVENT_NUMBER_IIC0_RXI
     (SYSTEM_IRQ_EVENT_NUMBER0)  /*!< Numbers 0/4/8/12/16/20/24/28 only */
#define SYSTEM_CFG_EVENT_NUMBER_CCC_PRD
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 0/4/8/12/16/20/24/28 only */
• • •
#define SYSTEM_CFG_EVENT_NUMBER_ADC140_WCMPUM
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 1/5/9/13/17/21/25/29 only */
#define SYSTEM_CFG_EVENT_NUMBER_IIC0_TXI
     (SYSTEM_IRQ_EVENT_NUMBER1)  /*!< Numbers 1/5/9/13/17/21/25/29 only */
#define SYSTEM_CFG_EVENT_NUMBER_DOC_DOPCI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 1/5/9/13/17/21/25/29 only */
• • •
#define SYSTEM_CFG_EVENT_NUMBER_ADC140_GCADI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 2/6/10/14/18/22/26/30 only */
#define SYSTEM_CFG_EVENT_NUMBER_IIC0_TEI
     (SYSTEM_IRQ_EVENT_NUMBER2)  /*!< Numbers 2/6/10/14/18/22/26/30 only */
#define SYSTEM_CFG_EVENT_NUMBER_CAC_MENDI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 2/6/10/14/18/22/26/30 only */
• • •
#define SYSTEM_CFG_EVENT_NUMBER_ACMP_CMPI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 3/7/11/15/19/23/27/31 only */
#define SYSTEM_CFG_EVENT_NUMBER_IIC0_EEI
     (SYSTEM_IRQ_EVENT_NUMBER3)  /*!< Numbers 3/7/11/15/19/23/27/31 only */
#define SYSTEM_CFG_EVENT_NUMBER_CAC_OVFI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 3/7/11/15/19/23/27/31 only */
```

Figure 2-8 Example of registering an interrupt to NVIC in r_system_cfg.h (Using RIIC0)

2.5 Macro and Type Definitions

For the I2C driver, the macro and types that can be referenced by the user are defined in the Driver_I2C.h file.

2.5.1 I2C Control Code Definitions

I2C control codes are I2C control commands to be used by the Control function.

Table 2-6 I2C Control Code List

Definition	Value	Description
ARM_I2C_OWN_ADDRESS	(0x01)	I2C's own slave address setting command
ARM_I2C_BUS_SPEED	(0x02)	I2C bus speed setting command
ARM_I2C_BUS_CLEAR	(0x03)	Bus clear command
ARM_I2C_ABORT_TRANSFER	(0x04)	Transmission/reception abort command

2.5.2 I2C Bus Speed Definitions

These are the definitions of the speed to be specified with the I2C bus speed setting command (ARM_I2C_BUS_SPEED).

Table 2-7 I2C Bus Speed Definition List

Definition	Value	Description
ARM_I2C_BUS_SPEED_STANDARD	(0x01)	Standard speed (100 kbps)
ARM_I2C_BUS_SPEED_FAST	(0x02)	Fast speed (400 kbps)
ARM_I2C_BUS_SPEED_FAST_PLUS	(0x03)	Disabled (Note)
ARM_I2C_BUS_SPEED_HIGH	(0x04)	Disabled (Note)

Note. This definition is not supported by the RE01 I2C driver. If it is specified by the Control function, ARM_DRIVER_ERROR_UNSUPPORTED will be returned.

2.5.3 I2C Address Flag Definitions

These are the definitions of options in setting a slave address.

Table 2-8 I2C Address Flag Definition List

Definition	Value	Description
ARM_I2C_ADDRESS_10BIT	(0x4000)	Unused (Note)
ARM_I2C_ADDRESS_GC	(0x8000)	General call definition In setting a slave address, specifying this value ORed with the own address will enable the response to a general call.

Note. This definition is not supported by this driver.

2.5.4 I2C Event Code Definitions

These are the definitions of events to be notified by callback functions. If multiple events occur at the same time, an ORed value will be notified.

Table 2-9 I2C Event Code List

Definition	Value	Description
ARM_I2C_EVENT_TRANSFER_DONE	(1UL << 0)	Communication was completed.
ARM_I2C_EVENT_TRANSFER_INCOMPLETE	(1UL << 1)	Communication was finished incompletely.
ARM_I2C_EVENT_SLAVE_TRANSMIT	(1UL << 2)	Slave transmission was started.
ARM_I2C_EVENT_SLAVE_RECEIVE	(1UL << 3)	Slave reception was started.
ARM_I2C_EVENT_ADDRESS_NACK	(1UL << 4)	NACK to the address was received from the slave.
ARM_I2C_EVENT_GENERAL_CALL	(1UL << 5)	A general call was accepted.
ARM_I2C_EVENT_ARBITRATION_LOST	(1UL << 6)	Arbitration lost occurred.
ARM_I2C_EVENT_BUS_ERROR	(1UL << 7)	Unused
ARM_I2C_EVENT_BUS_CLEAR	(1UL << 8)	Unused

2.6 Structure Definitions

For the I2C driver, the structures that can be referenced by the user are defined in the Driver_I2C.h file.

2.6.1 ARM_I2C_STATUS Structure

This structure is used when the GetStatus function returns the status of the I2C.

Table 2-10 ARM_I2C_STATUS Structure

Element Name	Type	Description
busy	uint32_t:1	Shows communication status. 0: Waiting for communication 1: Communication in progress (busy)
mode	uint32_t:1	Shows the master or slave mode. 0: Slave 1: Master
direction	uint32_t:1	Shows transmission or reception. 0: Transmission 1: Reception
general_call	uint32_t:1	Shows whether a general call was accepted. 0: General call not received 1: General call received
arbitration_lost	uint32_t:1	Shows whether arbitration lost was detected. 0: Arbitration lost not detected 1: Arbitration lost detected
bus_error	uint32_t:1	Unused (fixed at 0)
reserved	uint32_t:26	Reserved area

2.6.2 ARM_I2C_CAPABILITIES Structure

This structure is used when the GetCapabilities function returns the capabilities of I2C.

Table 2-11 ARM_I2C_CAPABILITIES Structure

Element Name	Type	Description
address_10_bit	uint32_t:1	Enable or disable of 10-bit address It always returns 0 (disabled).
reserved	uint32_t:31	Reserved area

2.7 State Transitions

The state transition diagram of the I2C driver is shown in Figure 2-9, and state-specific events and actions are shown in Table 2-12.

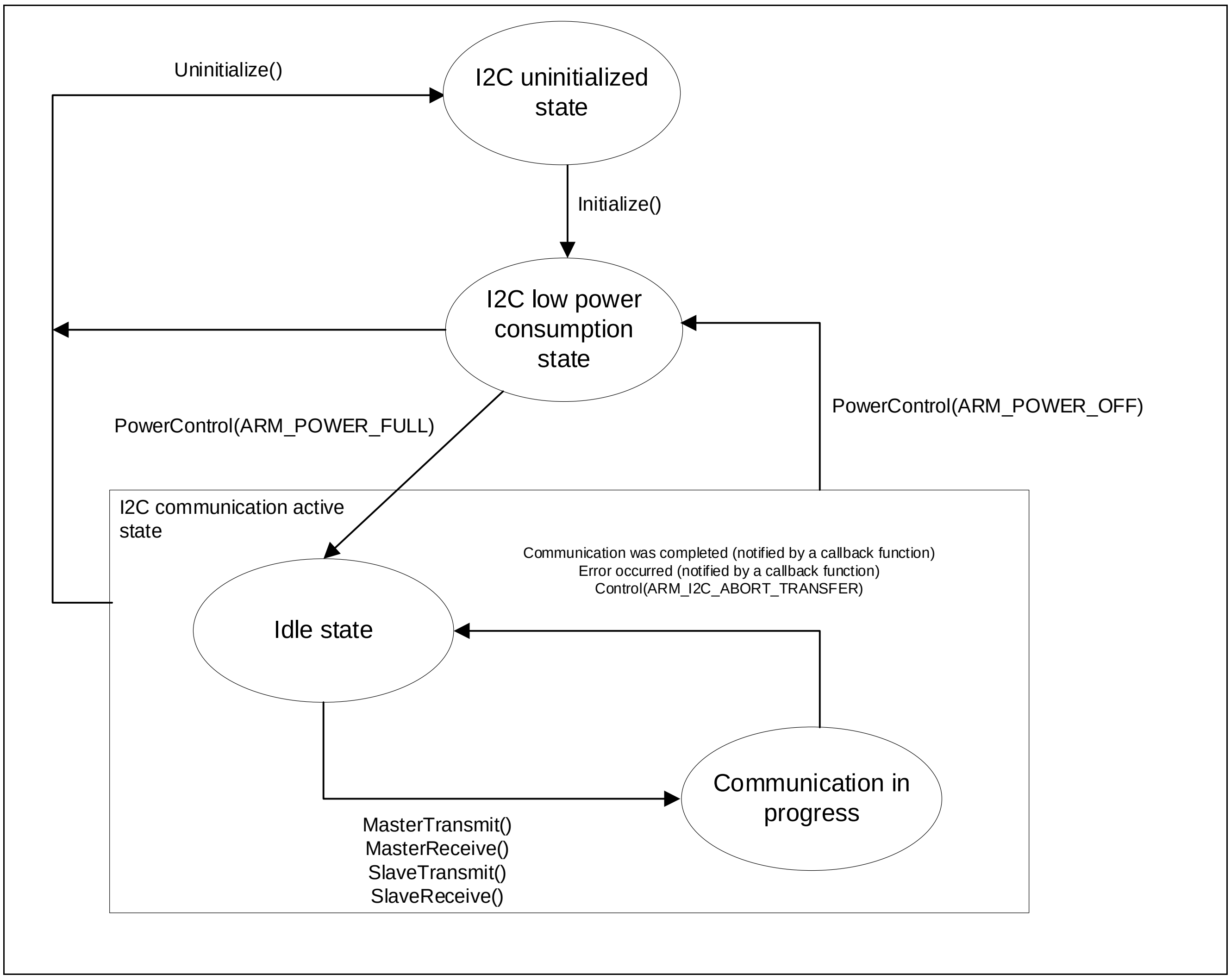

Uninitialize()
I2C uninitialized state
Initialize()
I2C low power consumption state
PowerControl(ARM_POWER_FULL)
PowerControl(ARM_POWER_OFF)
I2C communication active state
Communication was completed (notified by a callback function)
Error occurred (notified by a callback function)
Control(ARM_I2C_ABORT_TRANSFER)
Idle state
Communication in progress
MasterTransmit()
MasterReceive()
SlaveTransmit()
SlaveReceive()

Figure 2-9 State Transitions of I2C Driver

Table 2-12 Events and Actions Specific to I2C Driver State (Note 1)

State	Overview	Event	Action
I2C uninitialized state	The I2C driver is in this state after release from a reset.	Executing the Initialize() function	Enters the I2C low power consumption state
I2C low power consumption state	No clock is supplied to the I2C module in this state.	Executing the Uninitialize() function	Enters the I2C uninitialized state
		Executing the PowerControl(ARM_POWER_FULL) function	Enters the I2C communication active state (idle state)
I2C communication active state (idle state)	The I2C is waiting for communication in this state.	Executing the Uninitialize() function	Enters the I2C uninitialized state
		Executing the PowerControl(ARM_POWER_OFF) function	Enters the I2C low power consumption state
		Executing the MasterTransmit() function	Enters the state of having communication due to master transmission
		Executing the MasterReceive() function	Enters the state of having communication due to master reception
		Executing the SlaveTransmit() function	Enters the state of having communication due to slave transmission
		Executing the SlaveReceive() function	Enters the state of having communication due to slave reception
		Executing the Control(ARM_I2C_OWN_ADDRESS) function	Sets the I2C's own slave address.
		Executing the Control(ARM_I2C_BUS_SPEED) function	Sets a bus speed.
		Executing the Control(ARM_I2C_BUS_CLEAR) function	Executes bus clear processing.
I2C communication active state (communication in progress)	I2C communication is in progress in this state.	Executing the Uninitialize() function	Enters the I2C uninitialized state
		Executing the PowerControl(ARM_POWER_OFF) function	Enters the I2C low power consumption state
		Completion of communication	Causes a transition to an idle state and calls a callback function (Note 2).
		Error occurrence	Causes a transition to an idle state and calls a callback function (Note 2).
		Executing the Control(ARM_I2C_ABORT_TRANSFER) function	Aborts communication and causes a transition to an idle state.
		Executing the Control(ARM_I2C_BUS_CLEAR) function	Executes bus clear processing.

Note 1. The GetVersion, GetCapabilities, GetStatus, and GetDataCount functions can be executed in any state.

Note 2. Only if a callback function is specified when the Initialize function is executed, the callback function will be called.

3. Descriptions of Driver Operations

The I2C driver implements I2C communication capabilities. This chapter shows the procedure for calling the I2C driver.

3.1 Initial Settings of I2C

The procedure for making initial settings of I2C is shown in Figure 3-1.

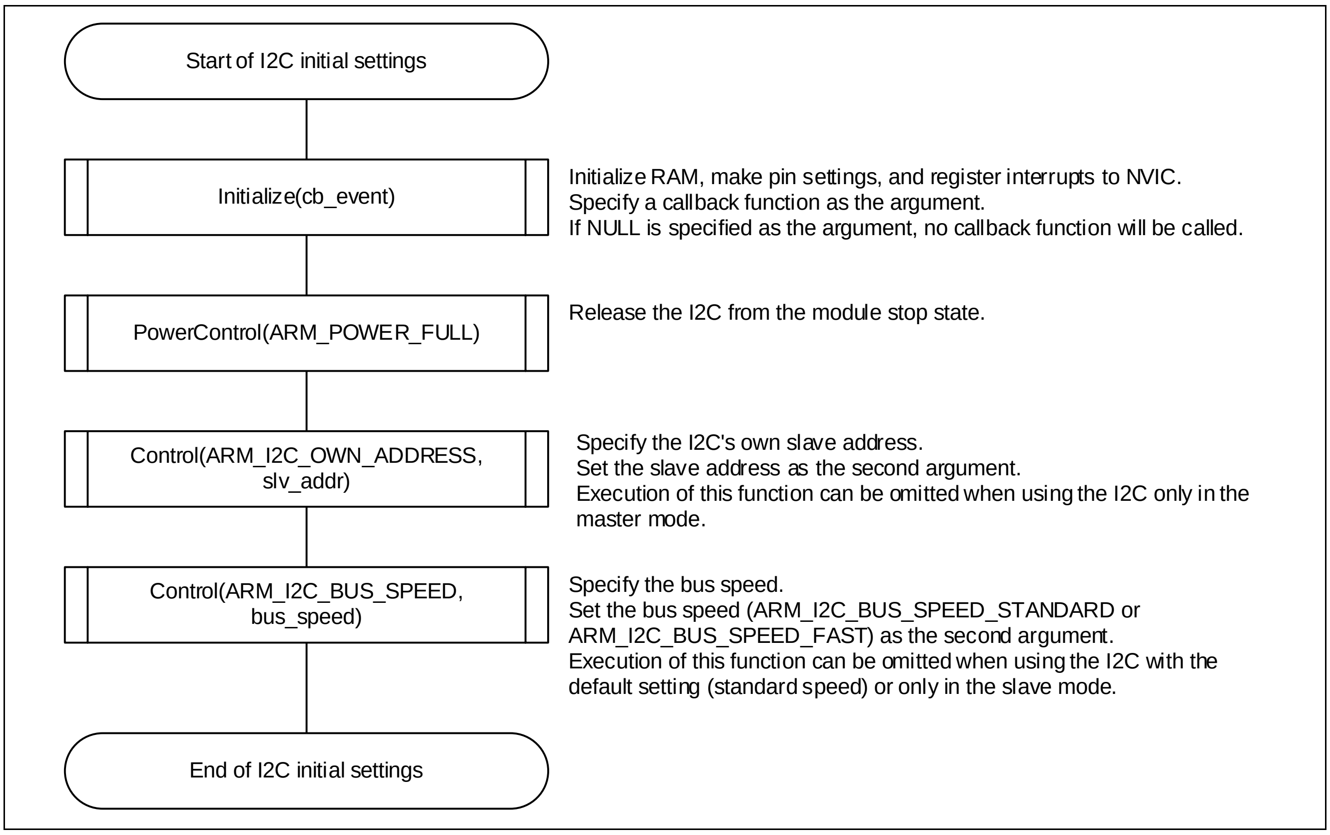

Start of I2C initial settings
Initialize(cb_event)
Initialize RAM, make pin settings, and register interrupts to NVIC.
Specify a callback function as the argument.
If NULL is specified as the argument, no callback function will be called.
PowerControl(ARM_POWER_FULL)
Release the I2C from the module stop state.
Control(ARM_I2C_OWN_ADDRESS, slv_addr)
Specify the I2C's own slave address.
Set the slave address as the second argument.
Execution of this function can be omitted when using the I2C only in the master mode.
Control(ARM_I2C_BUS_SPEED, bus_speed)
Specify the bus speed.
Set the bus speed (ARM_I2C_BUS_SPEED_STANDARD or ARM_I2C_BUS_SPEED_FAST) as the second argument.
Execution of this function can be omitted when using the I2C with the default setting (standard speed) or only in the slave mode.
End of I2C initial settings

Figure 3-1 I2C Initialization Procedure

3.2 Master Transmission

When master transmission is started, a start condition is output first, and then the address (W) signal of the transfer destination device and transmit data are output in this order. After all the data have been output with pending mode disabled, a stop condition is output to complete the master transmission.

With pending mode enabled, the stop condition is not output. If master transmission or master reception is executed again, the transfer will be started through a restart condition.

If RIIC_ADDR_NONE is specified as the address of the communication device, only a start condition and a stop condition will be output. (Even with pending mode enabled, a stop condition is output.)

The procedure for master transmission in the I2C is shown in Figure 3-2.

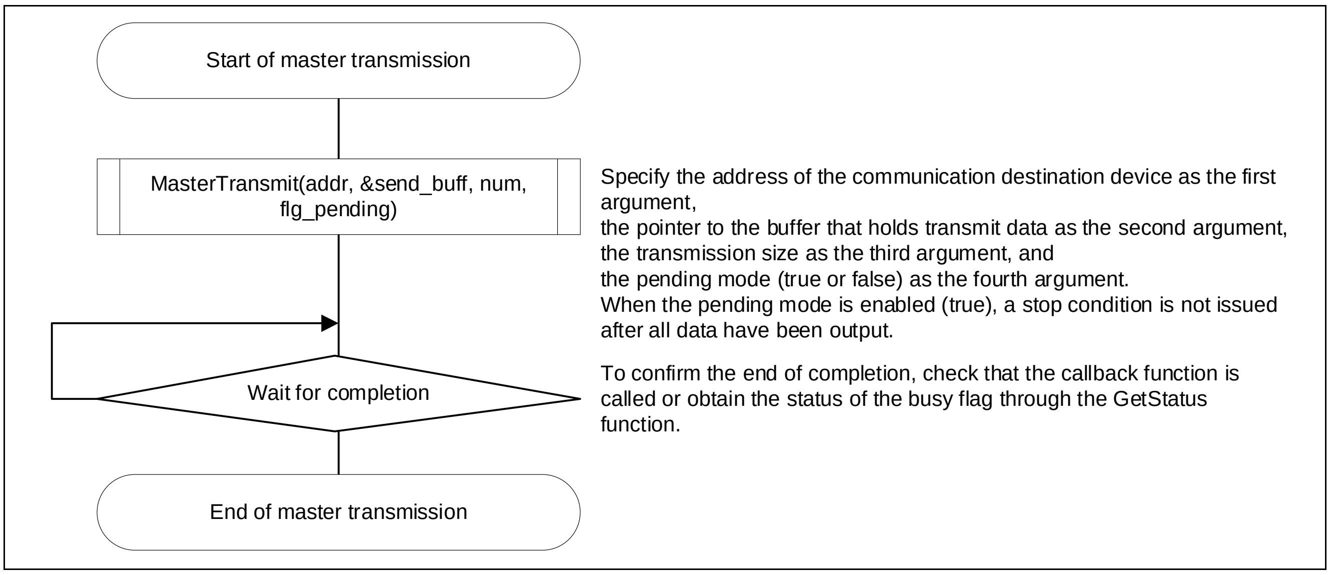

Start of master transmission
MasterTransmit(addr, &send_buff, num, flg_pending)
Specify the address of the communication destination device as the first argument,
the pointer to the buffer that holds transmit data as the second argument,
the transmission size as the third argument, and
the pending mode (true or false) as the fourth argument.
When the pending mode is enabled (true), a stop condition is not issued after all data have been output.
Wait for completion
To confirm the end of completion, check that the callback function is called or obtain the status of the busy flag through the GetStatus function.
End of master transmission

Figure 3-2 Master Transmit Procedure

If a callback function has been specified, it will be called with event information taken as an argument when master transmission is completed.

The event information to be generated in master transmission is shown in Table 3-1.

Table 3-1 Event Information Generated in Master Transmission

Event Information	Description
ARM_I2C_EVENT_TRANSFER_DONE	Master transmission was completed successfully.
ARM_I2C_EVENT_TRANSFER_DONE + ARM_I2C_EVENT_ADDRESS_NACK + ARM_I2C_EVENT_TRANSFER_INCOMPLETE	When an address was transmitted, NACK was received. (Note).
ARM_I2C_EVENT_TRANSFER_DONE + ARM_I2C_EVENT_TRANSFER_INCOMPLETE	When data was transmitted, NACK was received, and the transmission was completed. (Note)
ARM_I2C_EVENT_ARBITRATION_LOST + ARM_I2C_EVENT_TRANSFER_DONE + ARM_I2C_EVENT_TRANSFER_INCOMPLETE	Arbitration lost occurred.

Note. If NACK is received, a stop condition will be output even with pending mode enabled.

For master transmit processing, communication is implemented by using TXI interrupt processing, TEI interrupt processing, and EEI interrupt processing. The master transmission operation is shown in Figure 3-3.

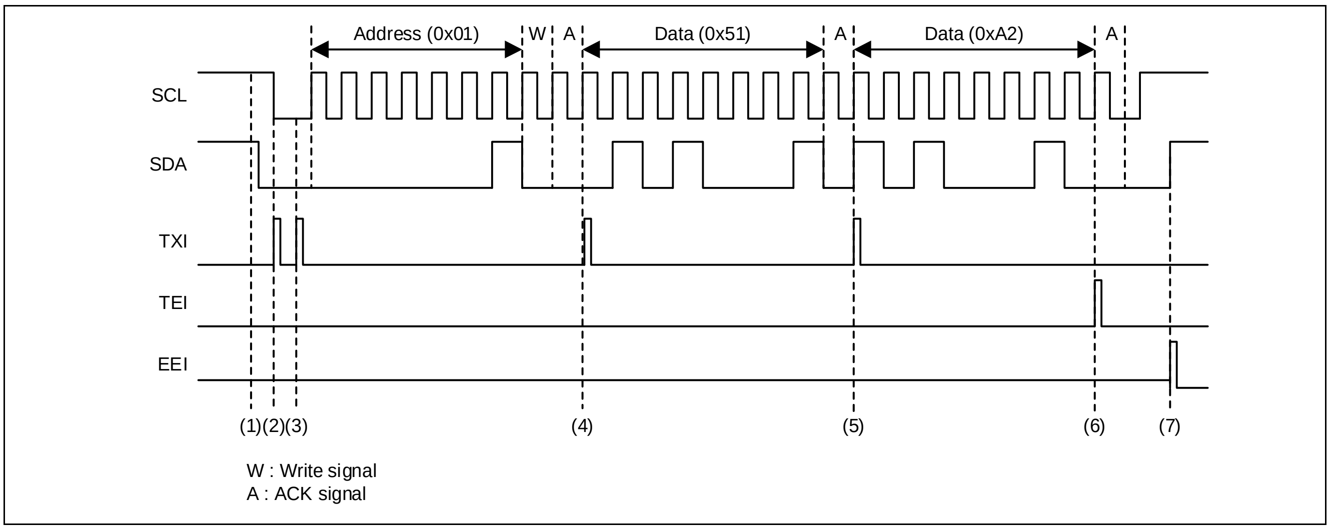


Figure 3-3 Master Transmission Operation (2-Byte Transmission)

(1) When the MasterTransmit function is executed, a start condition is output.
(2) By a TXI interrupt, address data is written to the transmit data register (ICDRT).
(3) By the second TXI interrupt, the first byte of transmit data is written to the ICDRT register.
(4) Every time a TXI interrupt occurs, transmit data is written to the ICDRT register one by one.
(5) By the TXI interrupt after the last data is written, TXI interrupts are disabled and TEI interrupts are enabled.
(6) After the last data is output, a TEI interrupt occurs and a stop condition is output.
(7) After the stop condition is output, an EEI interrupt occurs and the callback function is called.

Note 1. If NACK is received, an EEI interrupt will occur. The transmit processing will be aborted by the EEI interrupt and the callback function is called.

Note 2. If arbitration-lost occurs, the transmission will be aborted at that point and the callback function is called.

3.3 Master Reception

When master reception is started, a start condition is output first, the address (R) signal of the transfer destination device is output, and then data reception is started. After a specified size of data has been received, a stop condition is output to complete the master reception.

The procedure for master reception in the I2C is shown in Figure 3-4.

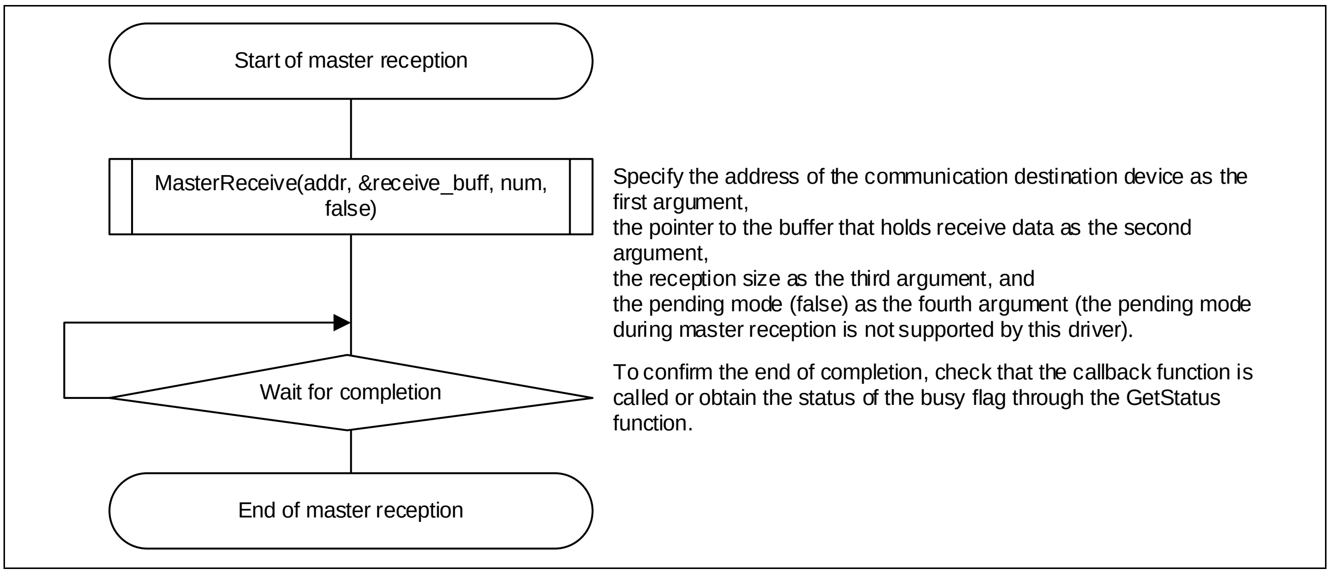

Start of master reception
MasterReceive(addr, &receive_buff, num, false)
Specify the address of the communication destination device as the first argument,
the pointer to the buffer that holds receive data as the second argument,
the reception size as the third argument, and
the pending mode (false) as the fourth argument (the pending mode during master reception is not supported by this driver).
Wait for completion
To confirm the end of completion, check that the callback function is called or obtain the status of the busy flag through the GetStatus function.
End of master reception

Figure 3-4 Master Receive Procedure

If a callback function has been specified, it will be called with event information taken as an argument when master reception is completed.

The event information to be generated in master reception is shown in Table 3-2.

Table 3-2 Event Information Generated in Master Reception

Event Information	Description
ARM_I2C_EVENT_TRANSFER_DONE	Master transmission was completed successfully.
ARM_I2C_EVENT_TRANSFER_DONE + ARM_I2C_EVENT_ADDRESS_NACK + ARM_I2C_EVENT_TRANSFER_INCOMPLETE	When an address was transmitted, NACK was received.
ARM_I2C_EVENT_ARBITRATION_LOST + ARM_I2C_EVENT_TRANSFER_DONE + ARM_I2C_EVENT_TRANSFER_INCOMPLETE	Arbitration lost occurred.

For master receive processing, communication is implemented by using TXI interrupt processing, RXI interrupt processing, and EEI interrupt processing. The master reception operation is shown in Figure 3-5.

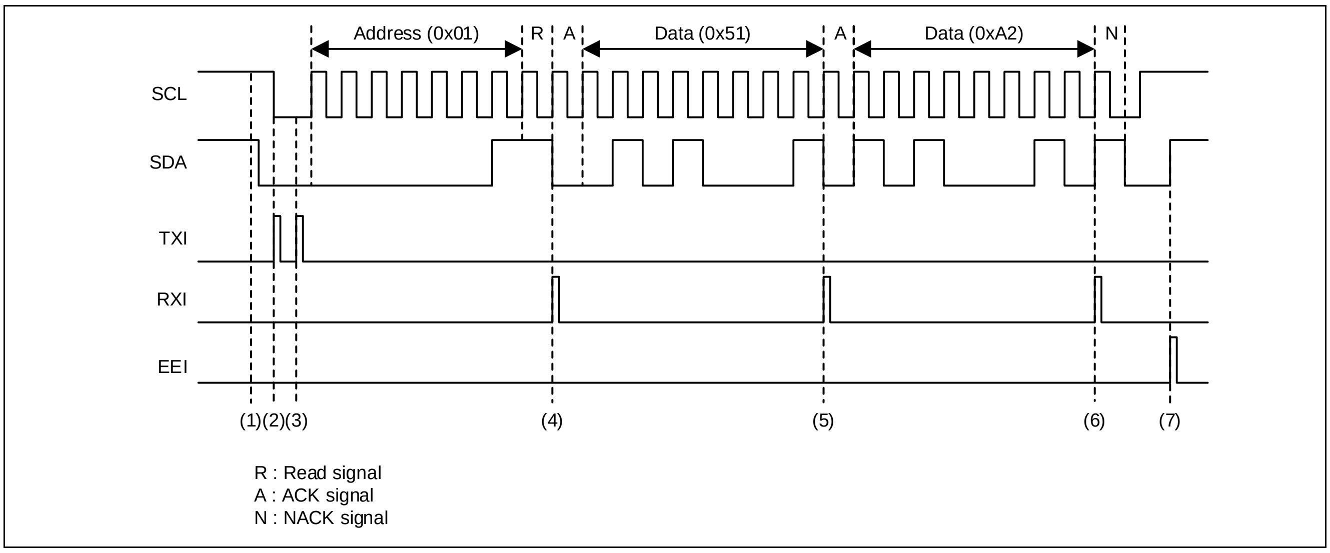


Figure 3-5 Master Reception Operation (2-Byte Reception)

(1) When the MasterReceive function is executed, a start condition is output.

(2) By a TXI interrupt, address data is written to the transmit data register (ICDRT).

(3) By the second TXI interrupt, the interrupt processing is finished without any processing.

(4) By an RXI interrupt due to a read signal, a dummy read from the receive data register (ICDRR) is performed.

(5) By the second RXI interrupt, the first byte of receive data is stored in a reception buffer.

(6) After the last data is received, the receive data is stored, NACK is returned, and then a stop condition is output.

(7) After the stop condition is output, an EEI interrupt occurs and the callback function is called.

Note 1. If NACK is received in address transfer, an EEI interrupt will occur. The receive processing will be aborted by the EEI interrupt and the callback function is called.

Note 2. If arbitration-lost occurs, the transmission will be aborted at that point and the callback function is called.

3.4 Slave Transmission

When slave transmission is started, the I2C enters the state of waiting for slave transmission. When a read signal for the I2C's own address is received from the master, the slave transmit operation is started.

The procedure for slave transmission in the I2C is shown in Figure 3-6.

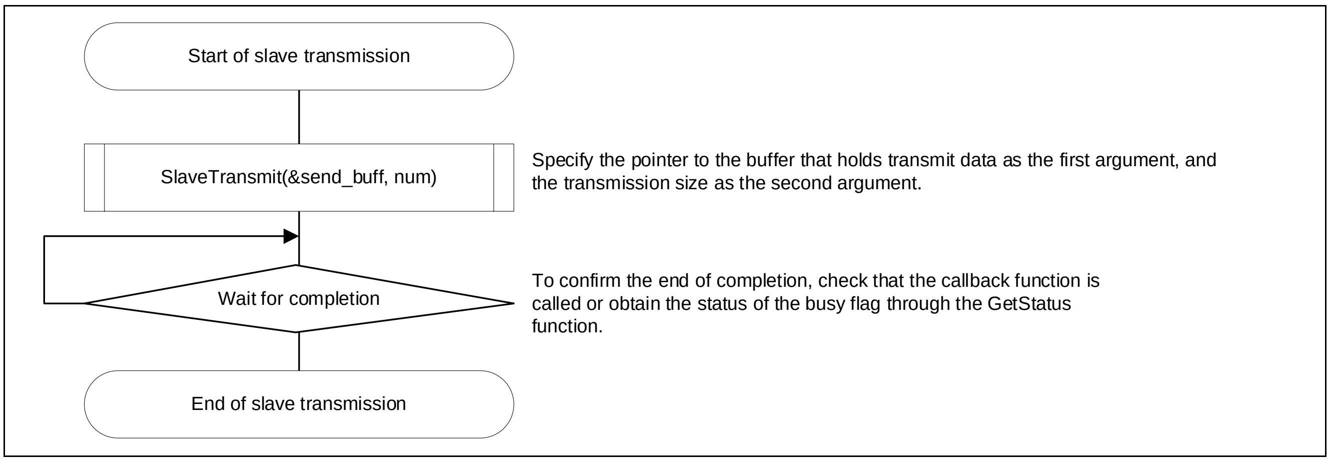

Start of slave transmission
SlaveTransmit(&send_buff, num)
Specify the pointer to the buffer that holds transmit data as the first argument, and the transmission size as the second argument.
Wait for completion
To confirm the end of completion, check that the callback function is called or obtain the status of the busy flag through the GetStatus function.
End of slave transmission

Figure 3-6 Slave Transmit Procedure

If a callback function has been specified, it will be called with event information taken as an argument when slave transmission is started or slave transmission is completed.

The event information to be generated in slave transmission is shown in Table 3-3.

Table 3-3 Event Information Generated in Slave Transmission

Event Information	Description
ARM_I2C_EVENT_SLAVE_TRANSMIT	Slave transmission was started.
ARM_I2C_EVENT_TRANSFER_DONE	Slave transmission was completed successfully.
ARM_I2C_EVENT_TRANSFER_DONE + ARM_I2C_EVENT_TRANSFER_INCOMPLETE	The transmission was not finished successfully. (Note)
ARM_I2C_EVENT_ARBITRATION_LOST + ARM_I2C_EVENT_TRANSFER_DONE + ARM_I2C_EVENT_TRANSFER_INCOMPLETE	Arbitration lost occurred.
ARM_I2C_EVENT_SLAVE_RECEIVE	A slave receive request was accepted.

Note. If one of the following conditions is fulfilled, the operation will be judged abnormal.
- If a stop condition is detected before transmission of a specified size of data
- If NACK is detected during data transmission
  (If a clock is supplied after NACK is detected, 0xFF will be transmitted.)
- If a W signal (receive request) is received during address reception
  (If reception is requested through a general call, ARM_I2C_EVENT_GENERAL_CALL will also be added to the event signal.)

For slave transmit processing, communication is implemented by using TXI interrupt processing, TEI interrupt processing, and EEI interrupt processing. The slave transmission operation is shown in Figure 3-7.

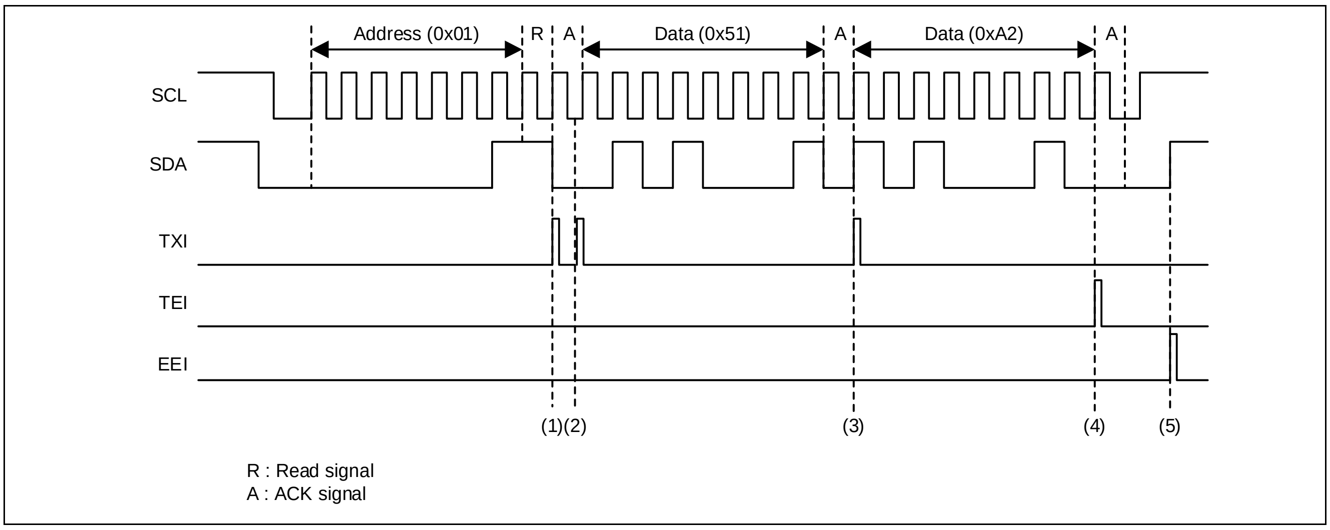


Figure 3-7 Slave Transmission Operation (2-Byte Transmission)

(1) Following the execution of the SlaveTransmit function, when the I2C's own address and a read signal are received, a TXI interrupt occurs and the first byte of transmit data is written to the transmit data register (ICDRT).
After that, the callback function is called.

(2) Every time a TXI interrupt occurs, transmit data is written to the ICDRT register one by one.

(3) By the TXI interrupt after the last data is written, TXI interrupts are disabled and TEI interrupts are enabled.

(4) After the last data is output, a TEI interrupt occurs.

(5) After a stop condition is output from the master, the callback function is called.

Note 1. If a W signal is received in address transfer, an RXI interrupt will occur and the callback function will be called. The subsequent receive data will only undergo dummy read, and an error will be returned by the callback function when a stop condition is detected.

Note 2. If NACK is received during data transmission, an EEI interrupt will occur and the transmit processing will be aborted. After that, even if a clock pulse is input, only 0xFF will be transmitted.

Note 3. If arbitration-lost occurs, the transmission will be aborted at that point and the callback function is called.

3.5 Slave Reception

When slave reception is started, the I2C enters the state of waiting for slave reception. When a write signal for the I2C's own address is received from the master, the slave receive operation is started.

The procedure for slave reception in I2C is shown in Figure 3-8.

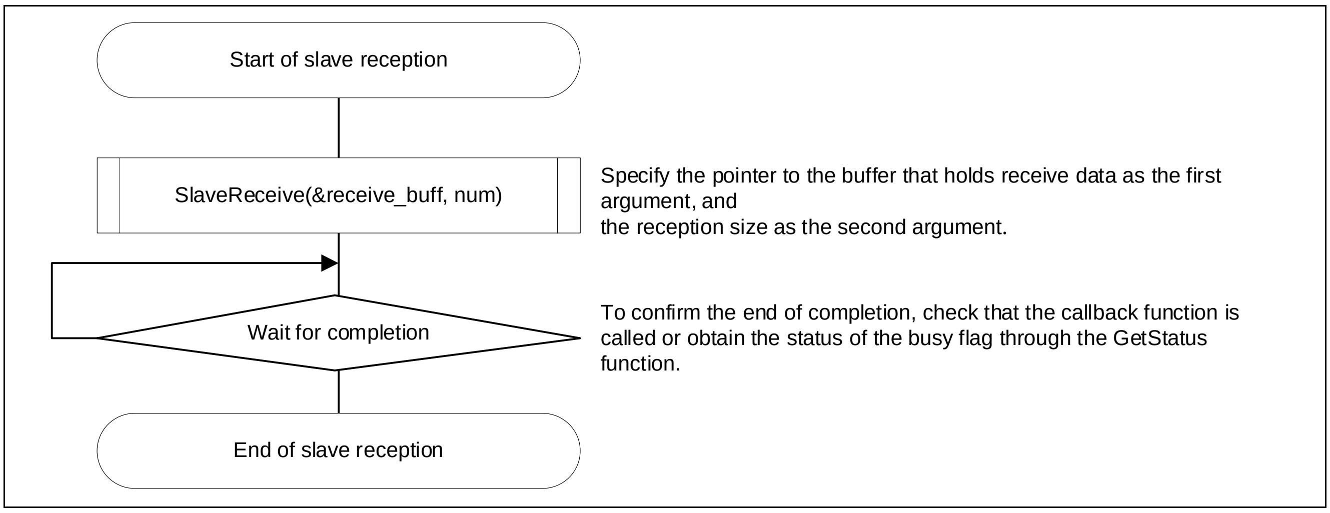

Start of slave reception
SlaveReceive(&receive_buff, num)
Specify the pointer to the buffer that holds receive data as the first argument, and
the reception size as the second argument.
Wait for completion
To confirm the end of completion, check that the callback function is called or obtain the status of the busy flag through the GetStatus function.
End of slave reception

Figure 3-8 Slave Receive Procedure

If a callback function has been specified, it will be called with event information taken as an argument when slave reception is started or slave reception is completed.

The event information to be generated in slave reception is shown in Table 3-4.

Table 3-4 Event Information Generated in Slave Reception

Event Information	Description
ARM_I2C_EVENT_SLAVE_RECEIVE	Slave reception was started.
ARM_I2C_EVENT_TRANSFER_DONE	Slave reception was completed successfully.
ARM_I2C_EVENT_TRANSFER_DONE+ ARM_I2C_EVENT_GENERAL_CALL	The reception due to a general call was completed successfully.
ARM_I2C_EVENT_TRANSFER_DONE + ARM_I2C_EVENT_TRANSFER_INCOMPLETE	The reception was not finished successfully. (Note)
ARM_I2C_EVENT_SLAVE_TRANSMIT	A slave transmit request was accepted.

Note. If one of the following conditions is fulfilled, the operation will be judged abnormal.
• If a stop condition is detected before reception of a specified size of data
(If the reception is started by a general call, ARM_I2C_EVENT_GENERAL_CALL will also be added to the event signal.)
• If an R signal (transmit request) is received during address reception

For slave receive processing, communication is implemented by using RXI interrupt processing and EEI interrupt processing. The slave reception operation is shown in Figure 3-9.

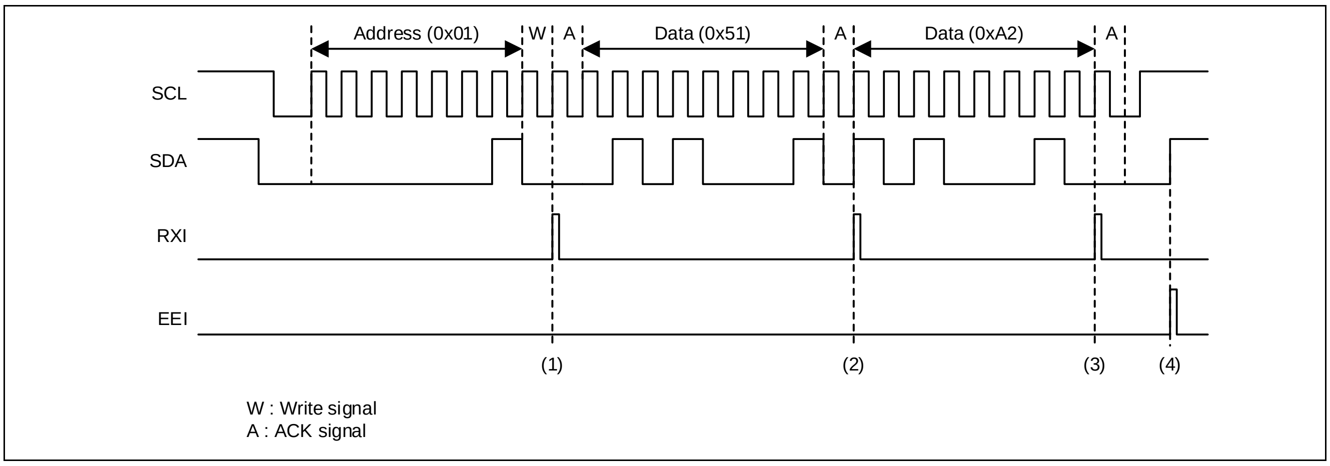


Figure 3-9 Slave Reception Operation (2-Byte Reception)

(1) Following the execution of the SlaveReceive function, when the I2C's own address and a write signal are received, an RXI interrupt occurs. (The first receive data is read but discarded.)
After that, the callback function is called.

(2) Every time an RXI interrupt occurs, receive data is stored in a specified reception buffer one by one.

(3) A NACK output setting is made by the RXI interrupt when the last data is received.
(After that, data received exceeding the specified size will be read but discarded, and NACK will be returned.)

(4) After a stop condition is output from the master, the callback function is called.

Note. If an R signal is received in address transfer, a TXI interrupt will occur and the callback function will be called. After that, if a clock pulse is input, only 0xFF will be transmitted, and an error will be returned by the callback function when a stop condition is detected.

3.6 Configurations

For the I2C driver, configuration items that can be specified by the user are provided in the r_i2c_cfg.h file.

3.6.1 Noise Filter Definition

This enables or disables a noise filter circuit and, if it is enabled, sets the number of stages in the noise filter.

Name: RIIC_NOISE_FILTER

Table 3-5 Settings of RIIC_NOISE_FILTER

Setting	Description
0	Disables the noise filter.
1	Filters out noise of up to 1IICφ cycle (single-stage filter)
2 (initial value)	Filters out noise of up to 2 IICφ cycles (2-stage filter)
3	Filters out noise of up to 3IICφ cycles (3-stage filter)
4	Filters out noise of up to 4IICφ cycles (4-stage filter)

3.6.2 Automatic Bus Speed Calculation Enable/Disable Definition

This enables or disables the automatic calculation of bus speed.

Name: RIIC_BUS_SPEED_CAL_ENABLE

Table 3-6 Settings of RIIC_BUS_SPEED_CAL_ENABLE

Setting	Description
0	Disables the automatic calculation of bus speed.
1 (initial value)	Enables the automatic calculation of bus speed.

The calculation provides flexibility, especially when clock rates change or are not typical values.

The drawback is this function requires linking of GCC floating point arithmetic library which increases code size. Please also refer to CHAPTER 5.

3.6.3 SCL Rise Time/Fall Time Definitions

These set the rise time and fall time of the SCL signal to be used in the automatic calculation of bus speed.

The time should be set in seconds.

Table 3-7 Names and Settings of SCL Rise Time/Fall Time Definitions

Name	Initial Value	Description
RIIC_STAD_SCL_UP_TIME	(1000E-9)	SCL signal rise time in standard mode (Initial value: 1000 ns)
RIIC_STAD_SCL_DOWN_TIME	(300E-9)	SCL signal fall time in standard mode (Initial value: 300 ns)
RIIC_FAST_SCL_UP_TIME	(300E-9)	SCL signal rise time in fast mode (Initial value: 300 ns)
RIIC_FAST_SCL_DOWN_TIME	(300E-9)	SCL signal fall time in fast mode (Initial value: 300 ns)

3.6.4 Bus Speed Setting Definitions

These define the register settings to be used when the automatic calculation of bus speed is disabled.

Table 3-8 Names and Settings of Bus Speed Setting Definition

Name	Initial Value	Description
RIIC_STAD_ICBRL	(15)	ICBRL setting in standard mode (0 to 31)
RIIC_STAD_ICBRH	(12)	ICBRH setting in standard mode (0 to 31)
RIIC_STAD_CKS	(3)	CKS setting in standard mode (0 to 7)
RIIC_FAST_ICBRL	(17)	ICBRL setting in fast mode (0 to 31)
RIIC_ FAST_ICBRH	(6)	ICBRH setting in fast mode (0 to 31)
RIIC_ FAST_CKS	(1)	CKS setting in fast mode (0 to 7)

3.6.5 TXI Interrupt Priority Level

This sets the priority level of the TXIn interrupt. (n = 0 or 1)

Name: RIIC0_TXI_PRIORITY, RIIC1_TXI_PRIORITY

Table 3-9 Settings of RIICn_TXI_PRIORITY

Setting	Description
0	Sets the interrupt priority level to 0. (highest priority)
1	Sets the interrupt priority level to 1.
2	Sets the interrupt priority level to 2.
3 (initial value)	Sets the interrupt priority level to 3.

3.6.6 TEI Interrupt Priority Level

This sets the priority level of the TEIn interrupt. (n = 0 or 1)

Name: RIIC0_TEI_PRIORITY, RIIC1_TEI_PRIORITY

Table 3-10 Settings of RIICn_TEI_PRIORITY

Setting	Description
0	Sets the interrupt priority level to 0. (highest priority)
1	Sets the interrupt priority level to 1.
2	Sets the interrupt priority level to 2.
3 (initial value)	Sets the interrupt priority level to 3.

3.6.7 RXI Interrupt Priority Level

This sets the priority level of the RXIn interrupt. (n = 0 or 1)

Name: RIIC0_RXI_PRIORITY, RIIC1_RXI_PRIORITY

Table 3-11 Settings of RIICn_RXI_PRIORITY

Setting	Description
0	Sets the interrupt priority level to 0. (highest priority)
1	Sets the interrupt priority level to 1.
2	Sets the interrupt priority level to 2.
3 (initial value)	Sets the interrupt priority level to 3.

3.6.8 EEI Interrupt Priority Level

This sets the priority level of the EEIn interrupt. (n = 0 or 1)

Name: RIIC0_EEI_PRIORITY, RIIC1_EEI_PRIORITY

Table 3-12 Settings of RIICn_EEI_PRIORITY

Setting	Description
0	Sets the interrupt priority level to 0. (highest priority)
1	Sets the interrupt priority level to 1.
2	Sets the interrupt priority level to 2.
3 (initial value)	Sets the interrupt priority level to 3.

3.6.9 Function Allocation to RAM

Make the setting for executing specific functions of the I2C driver via RAM.

This configuration item for setting function allocation to RAM has function-specific definitions.

Name: RIIC_CFG_SECTION_xxx

A function name xxx should be written in all capital letters.

Example: ARM_I2C_INITIALIZE function → RIIC_CFG_SECTION_ARM_I2C_INITIALIZE

Table 3-13 Settings of RIIC_CFG_SECTION_xxx

Setting	Description
SYSTEM_SECTION_CODE	Does not allocate the function to RAM.
SYSTEM_SECTION_RAM_FUNC	Allocates the functions into RAM.

Table 3-14 Initial State of Function Allocation to RAM

No.	Function Name	Allocation to RAM
1	ARM_I2C_Initialize	
2	ARM_I2C_Uninitialize	
3	ARM_I2C_PowerControl	
4	ARM_I2C_MasterTransmit	
5	ARM_I2C_MasterReceive	
6	ARM_I2C_SlaveTransmit	
7	ARM_I2C_SlaveReceive	
8	ARM_I2C_GetDataCount	
9	ARM_I2C_Control	
10	ARM_I2C_GetStatus	
11	ARM_I2C_GetVersion	
12	ARM_I2C_GetCapabilities	
13	iic_txi_interrupt (TXI interrupt processing)	✔
14	iic_tei_interrupt (TEI interrupt processing)	✔
15	iic_rxi_interrupt (RXI interrupt processing)	✔
16	iic_eei_interrupt (EEI interrupt processing)	✔

4. Detailed Information of Driver

This chapter describes the detailed specifications implementing the capabilities of this driver.

4.1 Function Specifications

The specifications and processing flow of each function of the I2C driver are described in this chapter.

Judgment on conditional branches is not always made as that described in the processing flow.

This chapter uses the following abbreviations.

ST: Start condition

SP: Stop condition

RS: Restart condition

4.1.1 ARM_I2C_Initialize Function

Table 4-1 ARM_I2C_Initialize Function Specifications

Format	static int32_t ARM_I2C_Initialize(ARM_I2C_SignalEvent_t cb_event, st_i2c_resources_t *p_i2c)	
Description	Initializes the I2C driver (initializes RAM, makes register settings, makes pin settings, and registers interrupts to NVIC). In the initial state, initializes RIIC at a communication speed of 100 kbps.	
Argument	ARM_I2C_SignalEvent_t cb_event: Callback function Specify the callback function to be executed when an event occurs. If NULL is set, the callback function will not be executed.	
	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C to initialize.	
Return value	ARM_DRIVER_OK	I2C initialization succeeded
	ARM_DRIVER_ERROR	I2C initialization failed If one of the following conditions is detected, the initialization will fail. • If the TXI, RXI, TEI, or EEI interrupt is defined as unused (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED) in r_system_cfg.h • If the setting of RIICn_TXI_PRIORITY has exceeded the definition range in r_i2c_cfg.h • If the setting of RIICn_RXI_PRIORITY has exceeded the definition range in r_i2c_cfg.h • If the setting of RIICn_TEI_PRIORITY has exceeded the definition range in r_i2c_cfg.h • If the setting of RIICn_EEI_PRIORITY has exceeded the definition range in r_i2c_cfg.h • If the resources of an RIIC channel to use is locked (If RIICn is already locked by the R_SYS_ResourceLock function)

Remarks	When this function is accessed, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	`static void callback(uint32_t event);`
	
	`// I2C driver instance ( RIIC0 )`
	`extern ARM_DRIVER_I2C Driver_I2C0;`
	`ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;`
	
	`main()`
	`{`
	`    I2cDev0->Initialize(callback);`
	`}`	

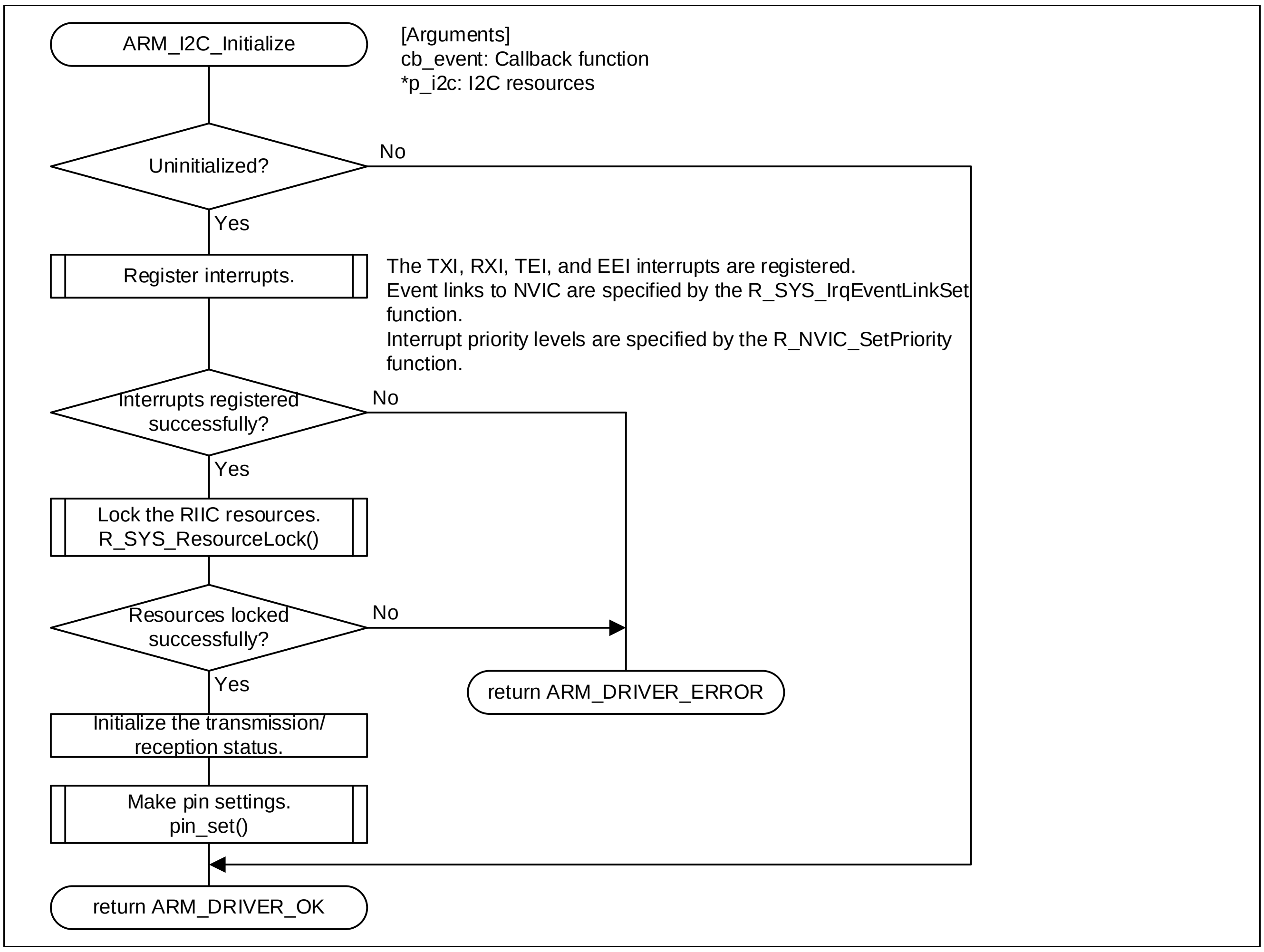

ARM_I2C_Initialize
[Arguments]
cb_event: Callback function
*p_i2c: I2C resources
Uninitialized?
No
Yes
Register interrupts.
The TXI, RXI, TEI, and EEI interrupts are registered.
Event links to NVIC are specified by the R_SYS_IrqEventLinkSet function.
Interrupt priority levels are specified by the R_NVIC_SetPriority function.
Interrupts registered successfully?
No
Yes
Lock the RIIC resources. R_SYS_ResourceLock()
Resources locked successfully?
No
Yes
return ARM_DRIVER_ERROR
Initialize the transmission/ reception status.
Make pin settings. pin_set()
return ARM_DRIVER_OK

Figure 4-1 ARM_I2C_Initialize Function Processing Flow

4.1.2 ARM_I2C_Uninitialize Function

Table 4-2 ARM_I2C_Uninitialize Function Specifications

Format	static int32_t ARM_I2C_Uninitialize(st_i2c_resources_t *p_i2c)
Description	Releases the I2C driver.
Argument	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C to release.
Return value	ARM_DRIVER_OK I2C release succeeded

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	// I2C driver instance (RIIC0)
	extern ARM_DRIVER_I2C Driver_I2C0;
	ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
	
	main()
	{
	I2cDev0->Uninitialize();
	}

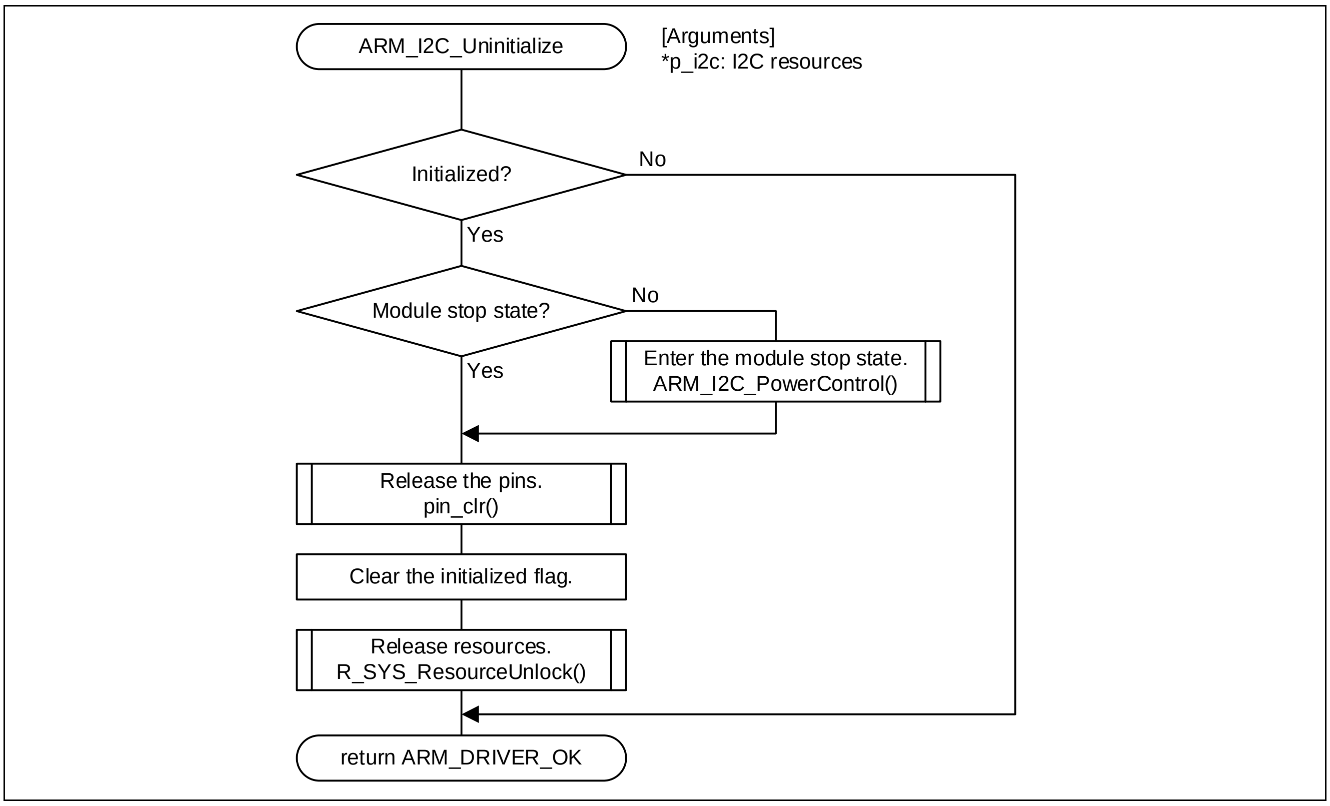


Figure 4-2 ARM_USART_Uninitialze Function Processing Flow

4.1.3 ARM_I2C_PowerControl Function

Table 4-3 ARM_I2C_PowerControl Function Specifications

Format	static int32_t ARM_I2C_PowerControl(ARM_POWER_STATE state, st_i2c_resources_t *p_i2c)	
Description	Releases the I2C from the module stop state or causes a transition to the mode.	
Argument	ARM_POWER_STATE state: Power setting Set one of the following. ARM_POWER_OFF: Causes a transition to the module stop state. ARM_POWER_FULL: Releases the I2C from the module stop state. ARM_POWER_LOW: This setting is not supported.	
	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C to supply power to.	
Return value	ARM_DRIVER_OK	Power setting change succeeded
	ARM_DRIVER_ERROR	Power setting change failed If one of the following conditions is detected, the power setting change will fail. • If this function is executed with I2C uninitialized • If transition to the module stop state has failed (If an error has occurred in R_LPM_ModuleStart) • If setting 100-kbps communication has failed (The setting will not fail if the I2C operates at not more than 32 MHz, which is the upper limit of PCLKB frequency.)
	ARM_DRIVER_ERROR_UNSUPPORTED	Unsupported power setting specified

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	`// I2C driver instance ( RIIC0 )`
	`extern ARM_DRIVER_I2C Driver_I2C0;`
	`ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;`
	
	`main()`
	`{`
	`    I2cDev0->Uninitialize();`
	`}`	

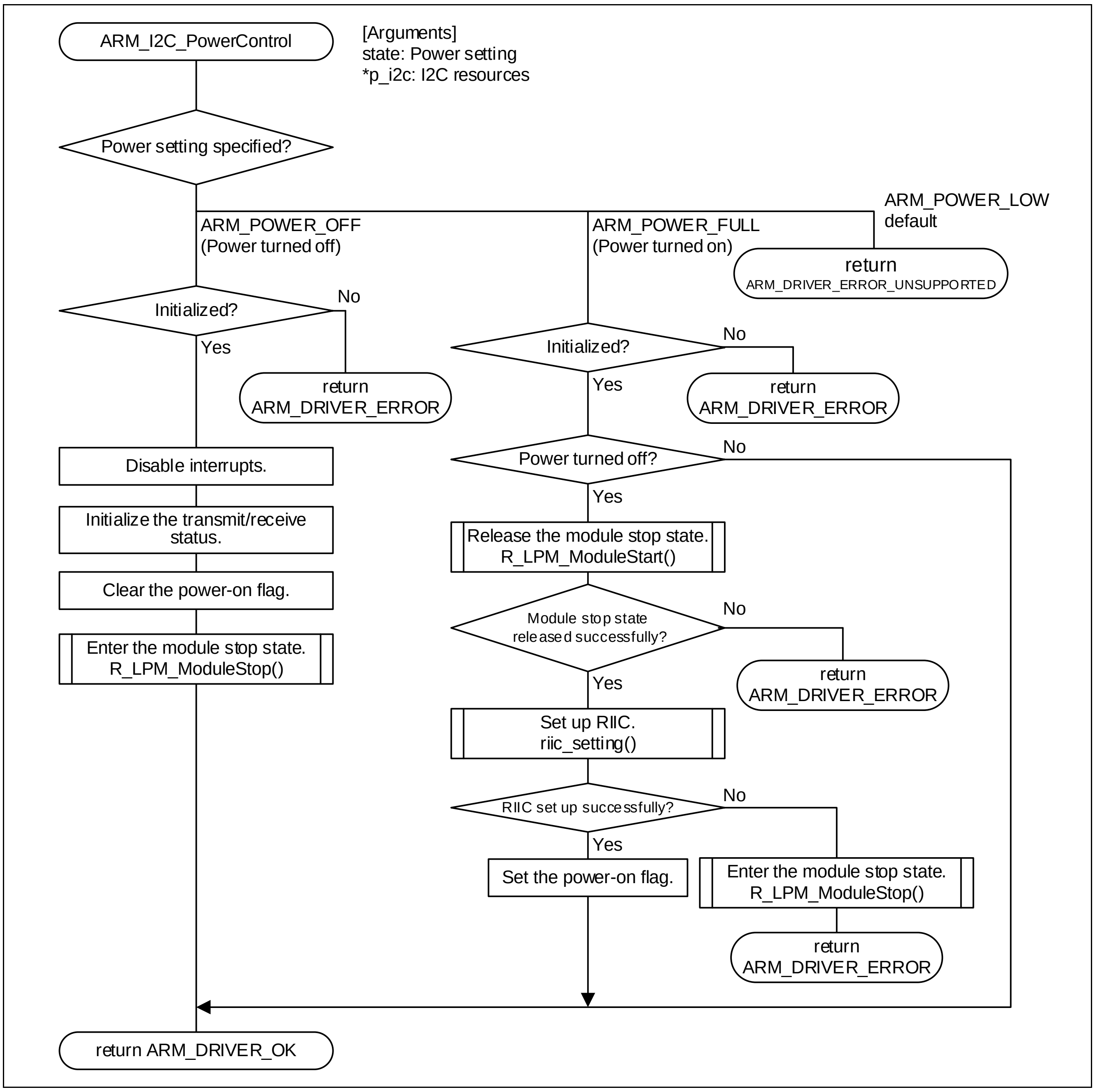


Figure 4-3 ARM_I2C_PowerControl Function Processing Flow

4.1.4 ARM_I2C_MasterTransmit Function

Table 4-4 ARM_I2C_MasterTransmit Function Specifications

Format	static int32_t ARM_I2C_MasterTransmit(uint32_t addr, const uint8_t *data, uint32_t num, bool xfer_pending, st_i2c_resources_t *p_i2c)	
Description	Starts master transmission.	
Argument	uint32_t addr: Address of transmission destination device If RIIC_ADDR_NONE is specified, only ST and SP will be output.	
	const uint8_t *data: Pointer to transmit data storage Specify the start address of the buffer where data to transmit is stored. Specify NULL when setting RIIC_ADDR_NONE as the address of the transmission destination device or the transmission size to 0. It is valid to set this pointer to NULL when “num”-parameter also is set to “0” ==> IIC-master will send IIC slave address with write-flag and wait for ACK & send STOP	
	uint32_t num: Transmission size Specify the size of data to transmit. If 0 is set, only ST, address, and SP will be transmitted.	
	bool xfer_pending: Setting of pending mode true: Pending mode enabled (After transmission is completed, a stop condition is not output.) false: Pending mode disabled (After transmission is completed, a stop condition is output.)	
	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C that transmit data.	
Return value	ARM_DRIVER_OK	Master transmission start succeeded
	ARM_DRIVER_ERROR Master transmission will fail if it is executed with the power supply turned off.	Master transmission failed
	ARM_DRIVER_ERROR_BUSY If one of the following conditions is detected, the transmission will fail because of a busy state. • Status judged as transmission in progress (status.busy == 1) • Judged as bus busy in slave state (ICCR2.MST=0, ICCR2.BBSY=1)	Transmission failed because of busy state
	ARM_DRIVER_ERROR_PARAMETER A parameter error will occur if the transmission size is 1 or greater and the transmission buffer is NULL.	Parameter error

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	`// I2C driver instance ( RIIC0 )`
	`extern ARM_DRIVER_I2C Driver_I2C0;`
	`ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;`
	`const uint8_t tx_data[2] = {0x51, 0xA2};`
	
	`main()`
	`{`
	`    I2cDev0->MasterTransmit(0x01, &tx_data[0], 2, false);`
	`}`	

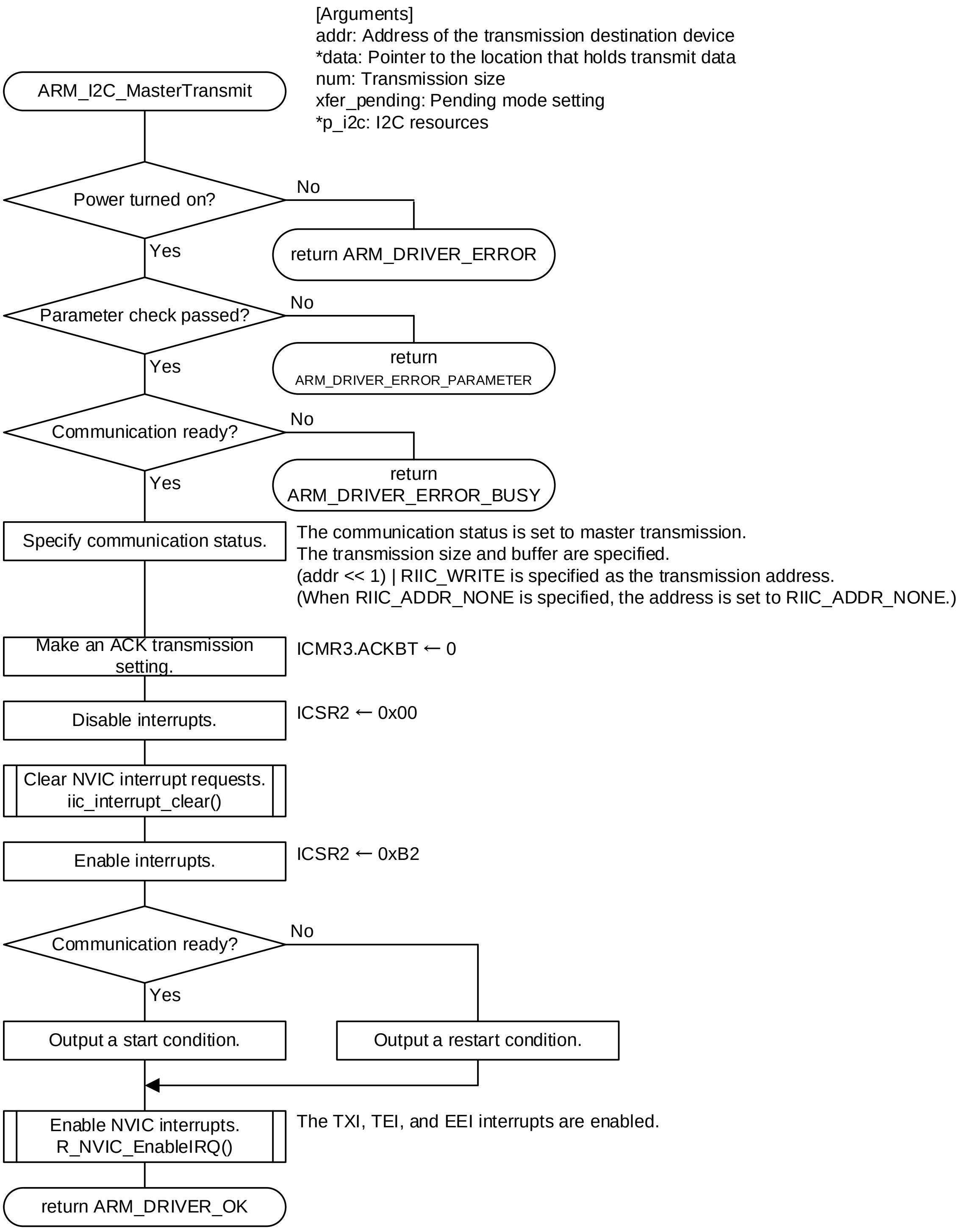

[Arguments]
addr: Address of the transmission destination device
*data: Pointer to the location that holds transmit data
num: Transmission size
xfer_pending: Pending mode setting
*p_i2c: I2C resources
ARM_I2C_MasterTransmit
Power turned on?
No
return ARM_DRIVER_ERROR
Yes
Parameter check passed?
No
return ARM_DRIVER_ERROR_PARAMETER
Yes
Communication ready?
No
return ARM_DRIVER_ERROR_BUSY
Yes
Specify communication status.
The communication status is set to master transmission.
The transmission size and buffer are specified.
(addr << 1) | RIIC_WRITE is specified as the transmission address.
(When RIIC_ADDR_NONE is specified, the address is set to RIIC_ADDR_NONE.)
Make an ACK transmission setting.
ICMR3.ACKBT ← 0
Disable interrupts.
ICSR2 ← 0x00
Clear NVIC interrupt requests. iic_interrupt_clear()
Enable interrupts.
ICSR2 ← 0xB2
Communication ready?
No
Yes
Output a start condition.
Output a restart condition.
Enable NVIC interrupts. R_NVIC_EnableIRQ()
The TXI, TEI, and EEI interrupts are enabled.
return ARM_DRIVER_OK

Figure 4-4 ARM_I2C_MasterTransmit Function Processing Flow

4.1.5 ARM_I2C_MasterReceive Function

Table 4-5 ARM_I2C_MasterReceive Function Specifications

Format	static int32_t ARM_I2C_MasterReceive(uint32_t addr, uint8_t *data, uint32_t num, bool xfer_pending, st_i2c_resources_t *p_i2c)
Description	Starts master reception.
Argument	uint32_t addr: Address of communication destination device
	const uint8_t *data: Pointer to receive data storage Specify the start address of the buffer where received data is to be stored.
	uint32_t num: Reception size Specify the size of data to receive.
	bool xfer_pending: Setting of pending mode (Only false is valid.) false: Pending mode disabled (After reception is completed, a stop condition is output.)
	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C that receive data.
Return value	ARM_DRIVER_OK Master reception start succeeded
	ARM_DRIVER_ERROR Master reception failed Master reception will fail if it is executed with the power supply turned off.
	ARM_DRIVER_ERROR_BUSY Reception failed because of busy state If one of the following conditions is detected, the reception will fail because of a busy state. • Status judged as transmission in progress (status.busy == 1) • Judged as bus busy in slave state (ICCR2.MST=0, ICCR2.BBSY=1)
	ARM_DRIVER_ERROR_PARAMETER Parameter error If one of the following conditions is detected, a parameter error will occur. • If true is specified in xfer_pending • If RIIC_ADDR_NONE is set as the communication destination device • If the pointer to receive data storage is NULL • If the reception size is 0

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	`// I2C driver instance ( RIIC0 )`
	`extern ARM_DRIVER_I2C Driver_I2C0;`
	`ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;`
	`uint8_t rx_data[2];`
	
	`main()`
	`{`
	`    I2cDev0->MasterReceive(0x01, &rx_data[0], 2, false);`
	`}`

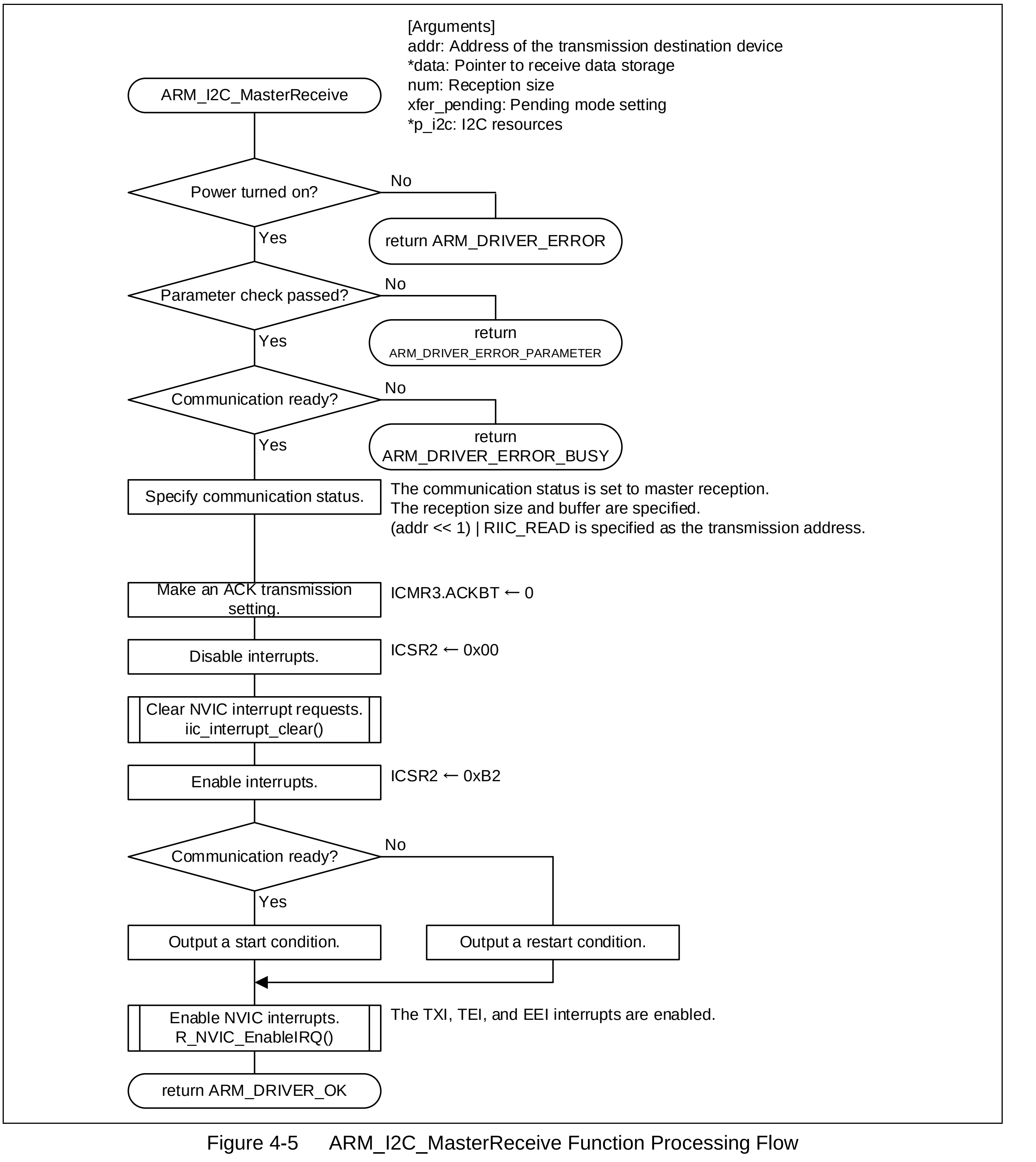

[Arguments]
addr: Address of the transmission destination device
*data: Pointer to receive data storage
num: Reception size
xfer_pending: Pending mode setting
*p_i2c: I2C resources
ARM_I2C_MasterReceive
Power turned on?
No
Yes
return ARM_DRIVER_ERROR
Parameter check passed?
No
Yes
return ARM_DRIVER_ERROR_PARAMETER
Communication ready?
No
Yes
return ARM_DRIVER_ERROR_BUSY
Specify communication status.
The communication status is set to master reception.
The reception size and buffer are specified.
(addr << 1) | RIIC_READ is specified as the transmission address.
Make an ACK transmission setting.
ICMR3.ACKBT ← 0
Disable interrupts.
ICSR2 ← 0x00
Clear NVIC interrupt requests. iic_interrupt_clear()
Enable interrupts.
ICSR2 ← 0xB2
Communication ready?
No
Yes
Output a start condition.
Output a restart condition.
Enable NVIC interrupts. R_NVIC_EnableIRQ()
The TXI, TEI, and EEI interrupts are enabled.
return ARM_DRIVER_OK

Figure 4-5 ARM_I2C_MasterReceive Function Processing Flow

4.1.6 ARM_I2C_SlaveTransmit Function

Table 4-6 ARM_I2C_SlaveTransmit Function Specifications

Format	static int32_t ARM_I2C_SlaveTransmit(const uint8_t *data, uint32_t num, st_i2c_resources_t *p_i2c)	
Description	Starts slave transmission.	
Argument	const uint8_t *data: Pointer to transmit data storage Specify the start address of the buffer where data to transmit is stored.	
	uint32_t num: Transmission size Specify the size of data to transmit.	
	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C that transmit data.	
Return value	ARM_DRIVER_OK	Slave transmission start succeeded
	ARM_DRIVER_ERROR Master transmission will fail if it is executed with the power supply turned off.	Slave transmission failed
	ARM_DRIVER_ERROR_BUSY If the status is judged as transmission in progress (status.busy == 1), the transmission will fail because of a busy state.	Transmission failed because of busy state
	ARM_DRIVER_ERROR_PARAMETER If one of the following conditions is detected, the a parameter error will occur. • If the transmission size is 0 • If the pointer to transmit data storage is NULL	Parameter error

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	// I2C driver instance (RIIC0)
	extern ARM_DRIVER_I2C Driver_I2C0;
	ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
	const uint8_t tx_data[2] = {0x51, 0xA2};
	
	main()
	{
	I2cDev0->SlaveTransmit(&tx_data[0], 2);
	}	

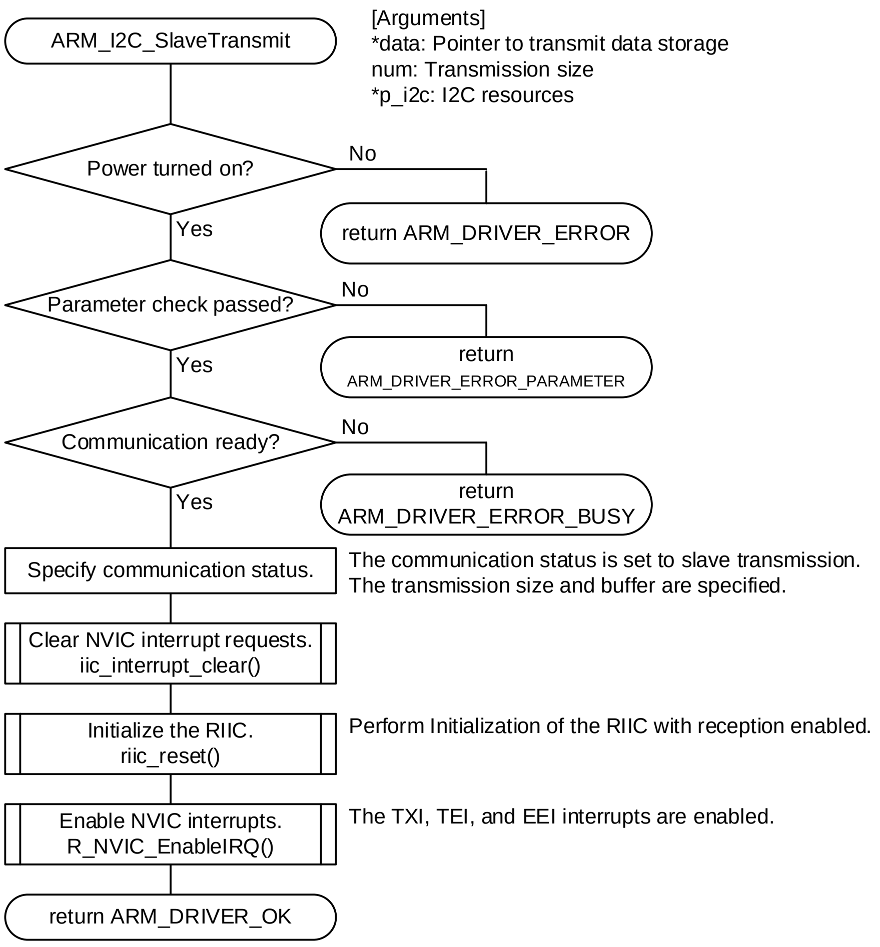

ARM_I2C_SlaveTransmit
[Arguments]
*data: Pointer to transmit data storage
num: Transmission size
*p_i2c: I2C resources
Power turned on?
No
Yes
return ARM_DRIVER_ERROR
Parameter check passed?
No
Yes
return ARM_DRIVER_ERROR_PARAMETER
Communication ready?
No
Yes
return ARM_DRIVER_ERROR_BUSY
Specify communication status.
The communication status is set to slave transmission. The transmission size and buffer are specified.
Clear NVIC interrupt requests. iic_interrupt_clear()
Initialize the RIIC. riic_reset()
Perform Initialization of the RIIC with reception enabled.
Enable NVIC interrupts. R_NVIC_EnableIRQ()
The TXI, TEI, and EEI interrupts are enabled.
return ARM_DRIVER_OK

Figure 4-6 ARM_I2C_SlaveTransmit Function Processing Flow

4.1.7 ARM_I2C_SlaveReceive Function

Table 4-7 ARM_I2C_SlaveReceive Function Specifications

Format	static int32_t ARM_I2C_SlaveReceive(uint8_t *data, uint32_t num, st_i2c_resources_t *p_i2c)	
Description	Starts slave reception.	
Argument	const uint8_t *data: Pointer to receive data storage Specify the start address of the buffer where received data is to be stored.	
	uint32_t num: Reception size Specify the size of data to receive.	
	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C that receive data.	
Return value	ARM_DRIVER_OK	Slave reception start succeeded
	ARM_DRIVER_ERROR Slave reception will fail if it is executed with the power supply turned off.	Slave reception failed
	ARM_DRIVER_ERROR_BUSY If the status is judged as transmission in progress (status.busy == 1), the reception will fail because of a busy state.	Reception failed because of busy state
	ARM_DRIVER_ERROR_PARAMETER If one of the following conditions is detected, a parameter error will occur. • If the pointer to receive data storage is NULL • If the reception size is 0	Parameter error

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	// I2C driver instance (RIIC0)
	extern ARM_DRIVER_I2C Driver_I2C0;
	ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
	uint8_t rx_data[2];
	
	main()
	{
	    I2cDev0->SlaveReceive(&rx_data[0], 2);
	}	

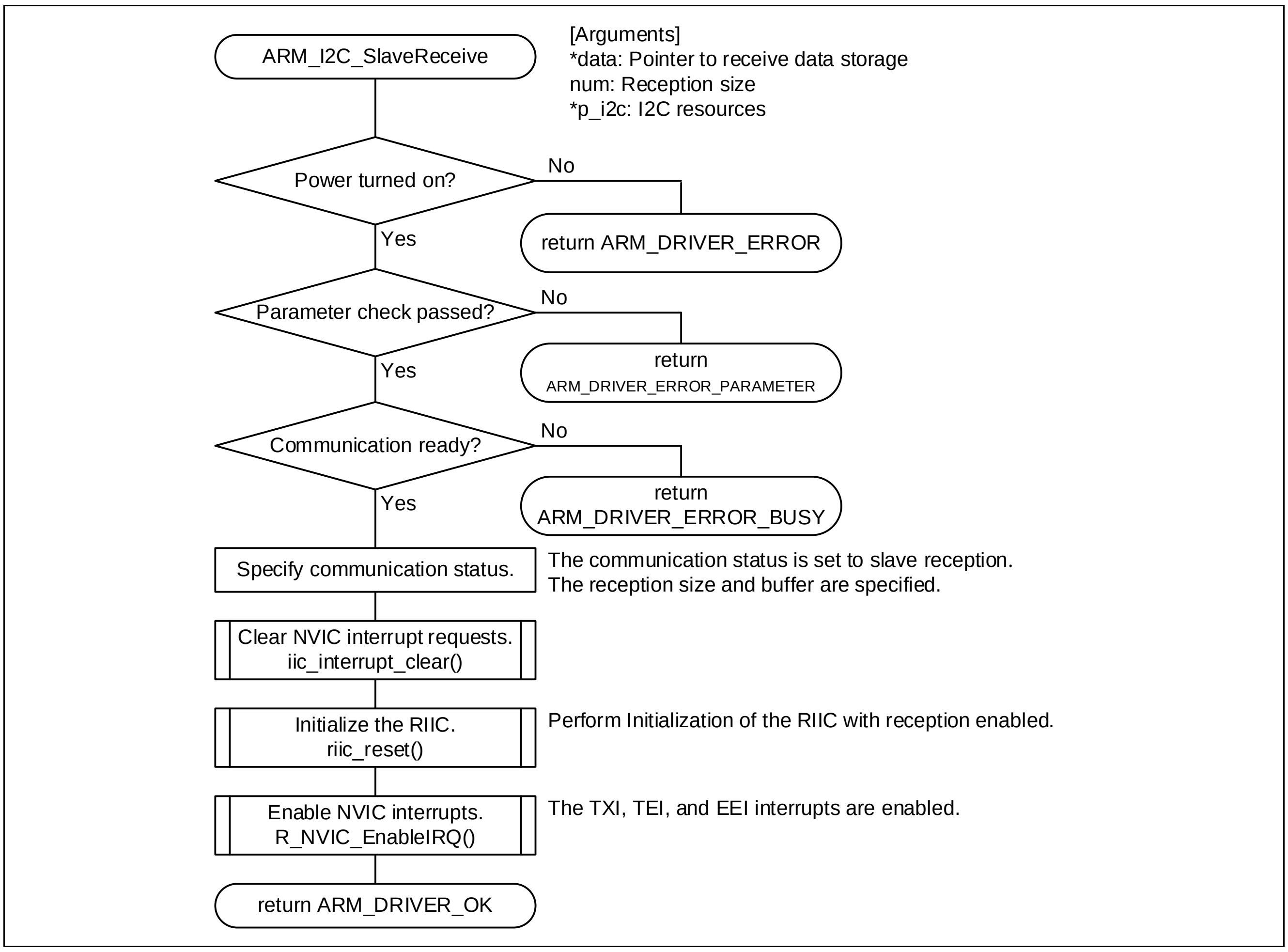

ARM_I2C_SlaveReceive
[Arguments]
*data: Pointer to receive data storage
num: Reception size
*p_i2c: I2C resources
Power turned on?
No
return ARM_DRIVER_ERROR
Yes
Parameter check passed?
No
return ARM_DRIVER_ERROR_PARAMETER
Yes
Communication ready?
No
return ARM_DRIVER_ERROR_BUSY
Yes
Specify communication status.
The communication status is set to slave reception.
The reception size and buffer are specified.
Clear NVIC interrupt requests. iic_interrupt_clear()
Initialize the RIIC. riic_reset()
Perform Initialization of the RIIC with reception enabled.
Enable NVIC interrupts. R_NVIC_EnableIRQ()
The TXI, TEI, and EEI interrupts are enabled.
return ARM_DRIVER_OK

Figure 4-7 ARM_I2C_SlaveReceive Function Processing Flow

4.1.8 ARM_I2C_GetDataCount Function

Table 4-8 ARM_I2C_GetDataCount Function Specifications

Format	static int32_t ARM_I2C_GetDataCount(st_i2c_resources_t *p_i2c)
Description	Obtains the transmit data count.
Argument	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C concerned.
Return value	Transmit data count

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	// I2C driver instance (RIIC0)
	extern ARM_DRIVER_I2C Driver_I2C0;
	ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
	
	main()
	{
	    uint32_t snd_cnt;
	    snd_cnt = I2cDev0->GetDataCount();
	}

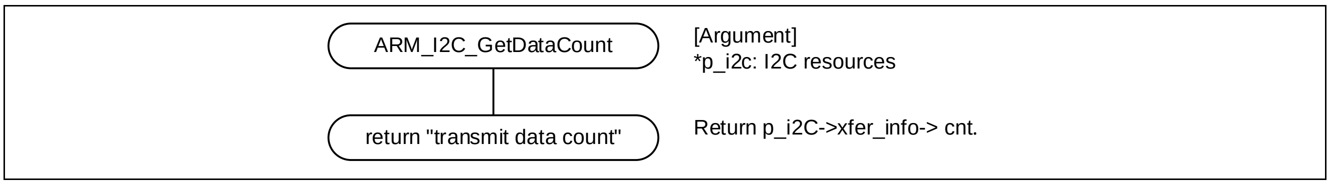


Figure 4-8 ARM_I2C_GetDataCount Function Processing Flow

4.1.9 ARM_I2C_Control Function

Table 4-9 ARM_I2C_Control Function Specifications

Format	static int32_t ARM_I2C_Control(uint32_t control, uint32_t arg, st_i2c_resources_t *p_i2c)	
Description	Executes a control command of the I2C.	
Argument	uint32_t control: Control command Specify one of the following control commands. • ARM_I2C_OWN_ADDRESS : I2C's own slave address setting command • ARM_I2C_BUS_SPEED : I2C bus speed setting command • ARM_I2C_BUS_CLEAR : Bus clear command • ARM_I2C_ABORT_TRANSFER : Transmission/reception abort command	
	uint32_t arg: Command-specific argument (See Table 4-10 for the relationship between control commands and arguments.)	
	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C to control.	
Return value	ARM_DRIVER_OK	Control command execution succeeded
	ARM_DRIVER_ERROR If one of the following conditions is detected, the control command execution will fail. Slave reception will fail if it is executed with the power supply turned off. • If the bus speed specified by the bus speed setting command was not able to be applied to actual operation.	Control command execution failed
	ARM_DRIVER_ERROR_BUSY If the I2C bus speed setting command (ARM_I2C_BUS_SPEED) is executed during communication, the control command execution will fail because of a busy state.	Control command execution failed because of busy state
	ARM_DRIVER_ERROR_UNSUPPORTED If one of the following conditions is detected, the control command execution will fail because of unsupported control. • If an out-of-spec value is specified as the control command • If an out-of-spec value is specified as the argument of the I2C bus speed setting command (ARM_I2C_BUS_SPEED)	Control command execution failed because of unsupported control

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	// I2C driver instance (RIIC0)
	extern ARM_DRIVER_I2C Driver_I2C0;
	ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
	
	main()
	{
	    I2cDev0->Control(ARM_I2C_OWN_ADDRESS, 0x01 | ARM_I2C_ADDRESS_GC);
	    I2cDev0->Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_FAST);
	}	

Table 4-10 Behaviors Specified with Control Commands and Command-Specific Arguments

Control Command (control)	Command-Specific Argument (arg)	Description
ARM_I2C_OWN_ADDRESS	0x00 to 0x7F (| ARM_I2C_ADDRESS_GC)	Sets the I2C's own slave address. If it is specified in combination with ARM_I2C_ADDRESS_GC, the I2C will also respond to a general call.
ARM_I2C_BUS_SPEED	ARM_I2C_BUS_SPEED_STANDARD	Sets the bus speed to the standard speed (100 kbps).
	ARM_I2C_BUS_SPEED_FAST	Sets the bus speed to the fast speed (400 kbps).
ARM_I2C_BUS_CLEAR	NULL	Executes the following processing, according to the current status. [In master mode] Outputs SCL for nine clock cycles. [In slave mode and with bus released] (1) Outputs ST. (2) Outputs SCL for nine clock cycles. (3) Outputs SP. [In slave mode and during transmission] Makes a data transmission of 0xFF. [In slave mode and during reception] Makes a NACK output setting.
ARM_I2C_ABORT_TRANSFER	NULL	Aborts transmission or reception.

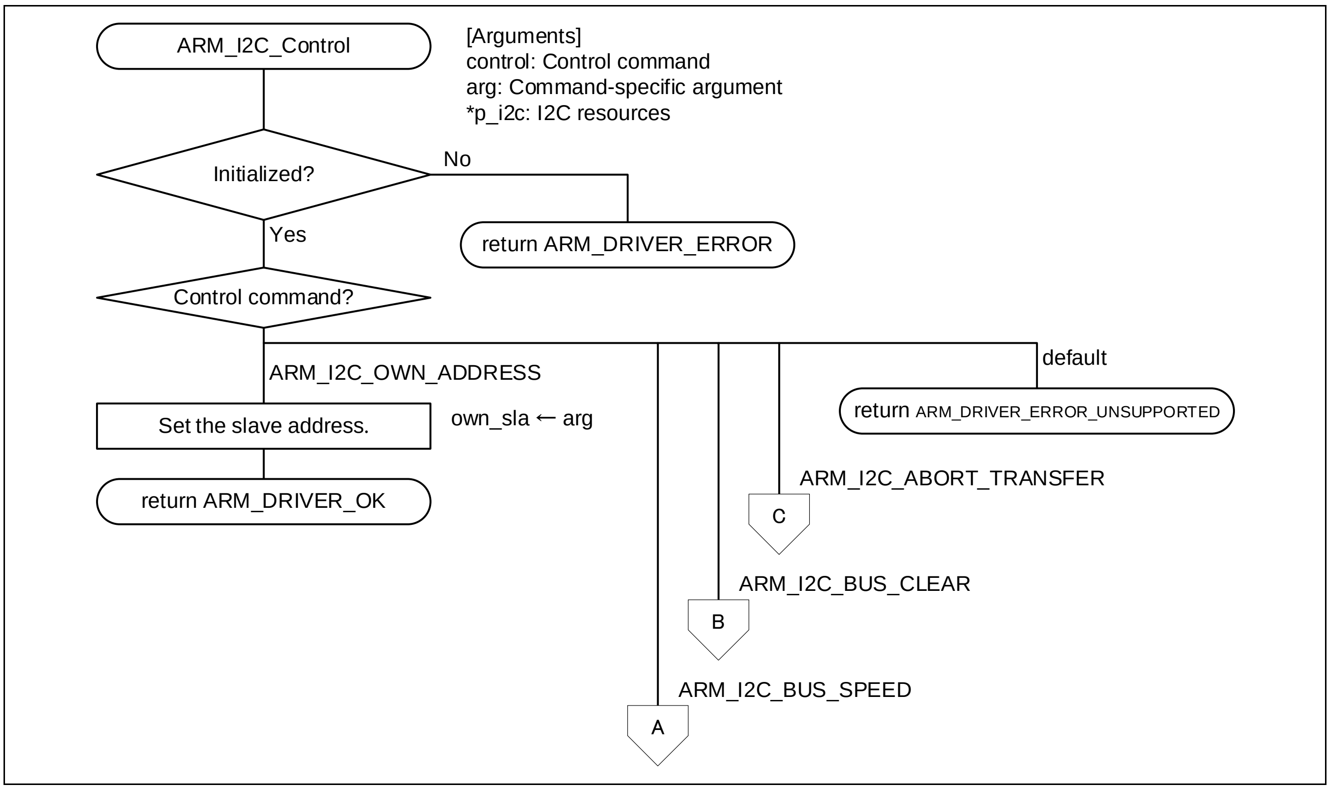


Figure 4-9 ARM_I2C_Control Function Processing Flow (1/3)

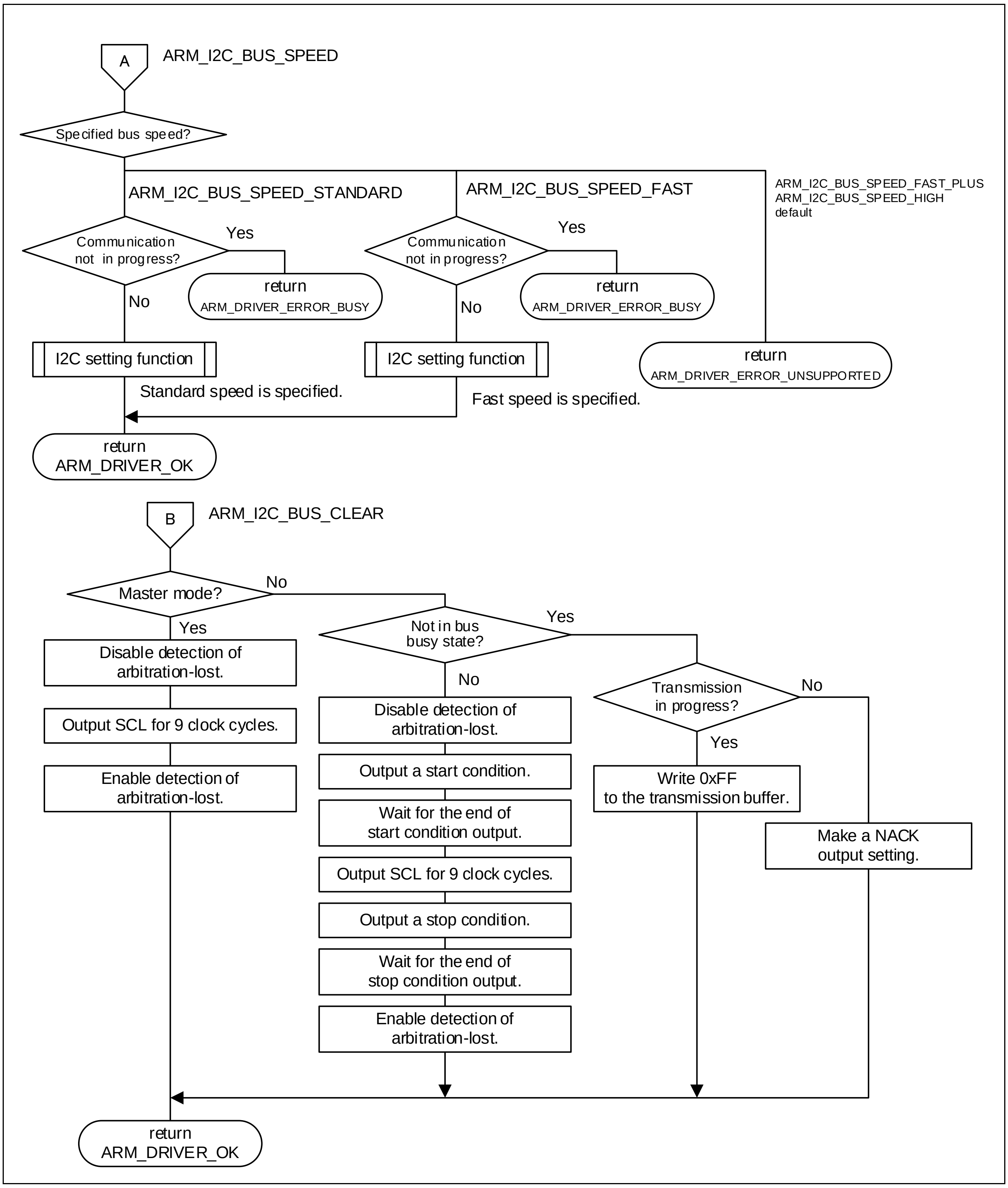

A
ARM_I2C_BUS_SPEED
Specified bus speed?
ARM_I2C_BUS_SPEED_STANDARD
ARM_I2C_BUS_SPEED_FAST
ARM_I2C_BUS_SPEED_FAST_PLUS
ARM_I2C_BUS_SPEED_HIGH
default
Communication not in progress?
Yes
return ARM_DRIVER_ERROR_BUSY
No
Communication not in progress?
Yes
return ARM_DRIVER_ERROR_BUSY
No
I2C setting function
I2C setting function
return ARM_DRIVER_ERROR_UNSUPPORTED
Standard speed is specified.
Fast speed is specified.
return ARM_DRIVER_OK
B
ARM_I2C_BUS_CLEAR
Master mode?
No
Yes
Disable detection of arbitration-lost.
Output SCL for 9 clock cycles.
Enable detection of arbitration-lost.
Not in bus busy state?
Yes
No
Disable detection of arbitration-lost.
Output a start condition.
Wait for the end of start condition output.
Output SCL for 9 clock cycles.
Output a stop condition.
Wait for the end of stop condition output.
Enable detection of arbitration-lost.
Transmission in progress?
No
Yes
Write 0xFF to the transmission buffer.
Make a NACK output setting.
return ARM_DRIVER_OK

Figure 4-10 ARM_I2C_Control Function Processing Flow (2/3)

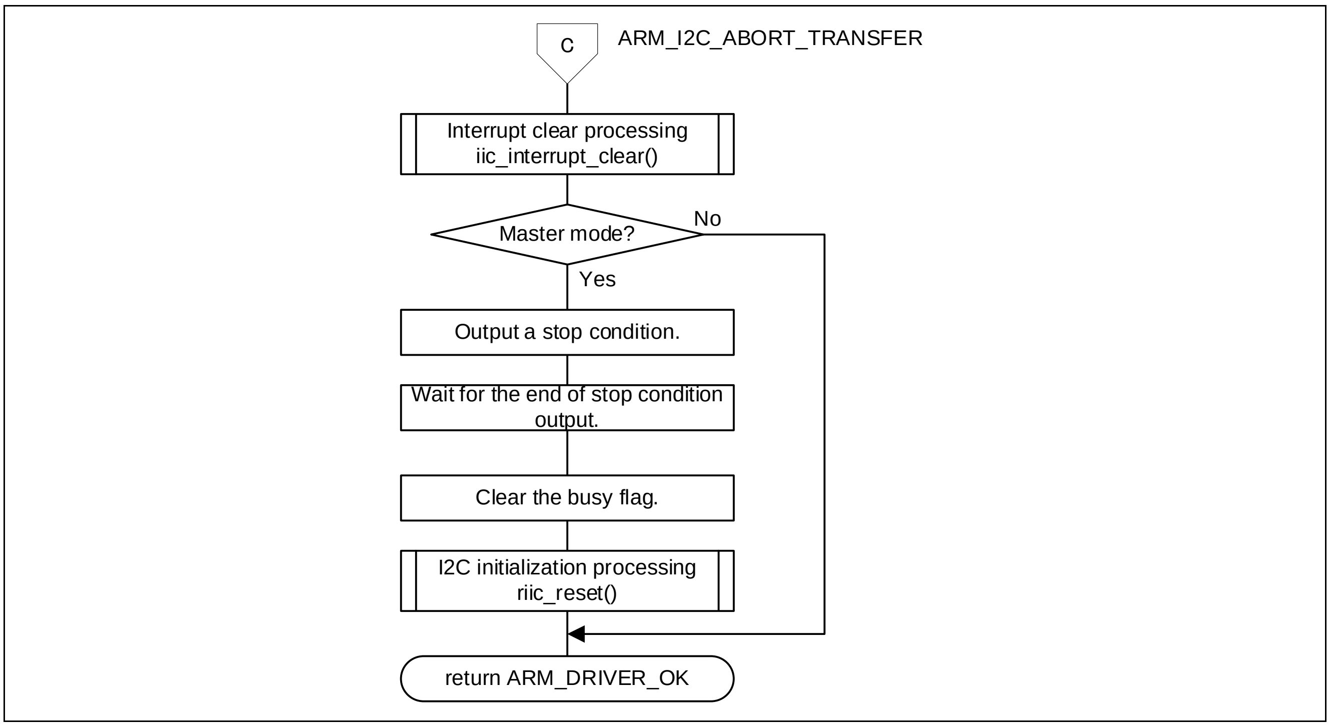


Figure 4-11 ARM_I2C_Control Function Processing Flow (3/3)

4.1.10 ARM_I2C_GetStatus Function

Table 4-11 ARM_I2C_GetStatus Function Specifications

Format	ARM_I2C_STATUS ARM_I2C_GetStatus(st_i2c_resources_t *p_i2c)
Description	Returns the status of I2C.
Argument	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C concerned.
Return value	Communication status

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	// I2C driver instance (RIIC0)
	extern ARM_DRIVER_I2C Driver_I2C0;
	ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
	
	main()
	{
	    ARM_I2C_STATUS state;
	    state = I2cDev0->GetStatus();
	
	}

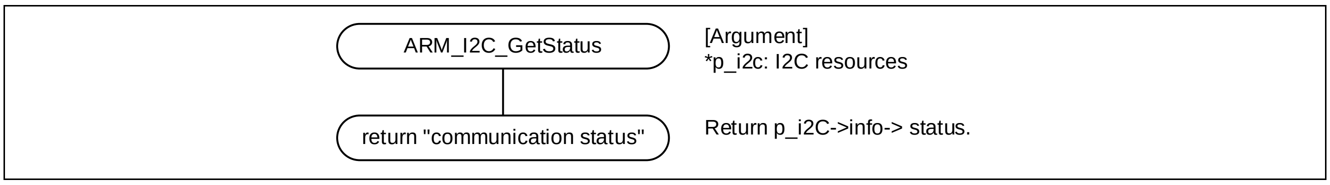


Figure 4-12 ARM_I2C_GetStatus Function Processing Flow

4.1.11 ARM_I2C_GetVersion Function

Table 4-12 ARM_I2C_GetVersion Function Specifications

Format	static ARM_DRIVER_VERSION ARM_I2C_GetVersion(void)
Description	Obtains the version of the I2C driver.
Argument	None
Return value	Version of the I2C driver

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	// I2C driver instance (RIIC0)
	extern ARM_DRIVER_I2C Driver_I2C0;
	ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
	
	main()
	{
	    ARM_DRIVER_VERSION version;
	    version = I2cDev0->GetVersion();
	
	}

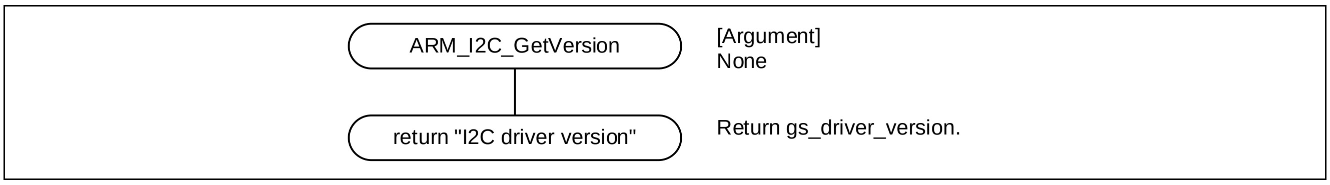


Figure 4-13 ARM_I2C_GetVersion Function Processing Flow

4.1.12 ARM_I2C_GetCapabilities Function

Table 4-13 ARM_I2C_ GetCapabilities Function Specifications

Format	static ARM_I2C_CAPABILITIES ARM_I2C_GetCapabilities(void)
Description	Obtains the capabilities of the I2C driver.
Argument	None
Return value	Driver capabilities

Remarks	When this function is accessed from the instance, specifying the I2C resources is not required.
	
	[Example of calling function from instance]
	// I2C driver instance (RIIC0)
	extern ARM_DRIVER_I2C Driver_I2C0;
	ARM_DRIVER_I2C *i2cDev0 = &Driver_I2C0;
	
	main()
	{
	    ARM_I2C_CAPABILITIES cap;
	    cap = I2cDev0->GetCapabilities();
	
	}

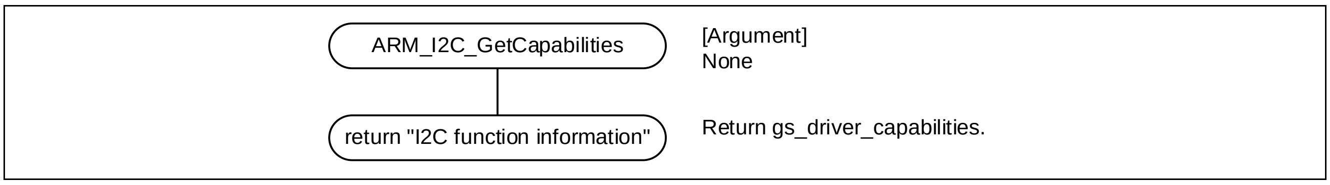

ARM_I2C_GetCapabilities
[Argument]
None
return "I2C function information"
Return gs_driver_capabilities.

Figure 4-14 ARM_I2C_GetCapabilities Function Processing Flow

4.1.13 riic_bps_calc Function

Table 4-14 riic_bps_calc Function Specifications

Format	static int32_t riic_bps_calc (uint16_t kbps, st_i2c_reg_buf_t *reg_val)	
Description	Calculates a bus speed.	
Argument	uint16_t kbps: Bus speed	
	st_i2c_reg_buf_t *reg_val: Buffer for setting registers Buffer for storing the result of bus speed calculation.	
Return value	ARM_DRIVER_OK	Bus speed calculation succeeded
	ARM_DRIVER_ERROR	Bus speed calculation failed
Remarks	The processing varies depending on RIIC_BUS_SPEED_CAL_ENABLE in r_i2c_cfg.h.	

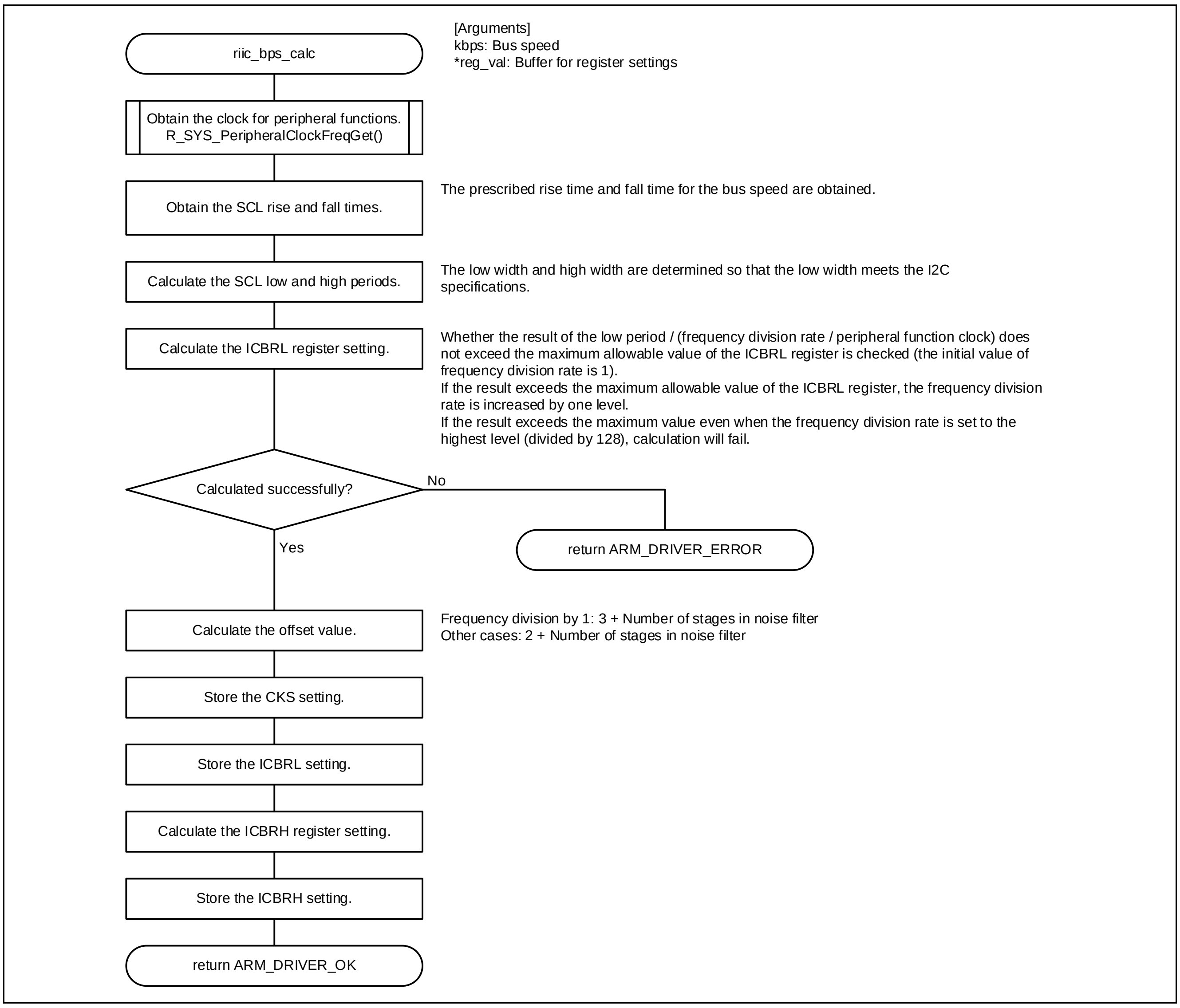

riic_bps_calc
[Arguments]
kbps: Bus speed
*reg_val: Buffer for register settings
Obtain the clock for peripheral functions. R_SYS_PeripheralClockFreqGet()
Obtain the SCL rise and fall times.
The prescribed rise time and fall time for the bus speed are obtained.
Calculate the SCL low and high periods.
The low width and high width are determined so that the low width meets the I2C specifications.
Calculate the ICBRL register setting.
Whether the result of the low period / (frequency division rate / peripheral function clock) does not exceed the maximum allowable value of the ICBRL register is checked (the initial value of frequency division rate is 1).
If the result exceeds the maximum allowable value of the ICBRL register, the frequency division rate is increased by one level.
If the result exceeds the maximum value even when the frequency division rate is set to the highest level (divided by 128), calculation will fail.
Calculated successfully?
No
return ARM_DRIVER_ERROR
Yes
Calculate the offset value.
Frequency division by 1: 3 + Number of stages in noise filter
Other cases: 2 + Number of stages in noise filter
Store the CKS setting.
Store the ICBRL setting.
Calculate the ICBRH register setting.
Store the ICBRH setting.
return ARM_DRIVER_OK

Figure 4-15 riic_bps_calc Function Processing Flow (for RIIC_BUS_SPEED_CAL_ENABLE = 1)

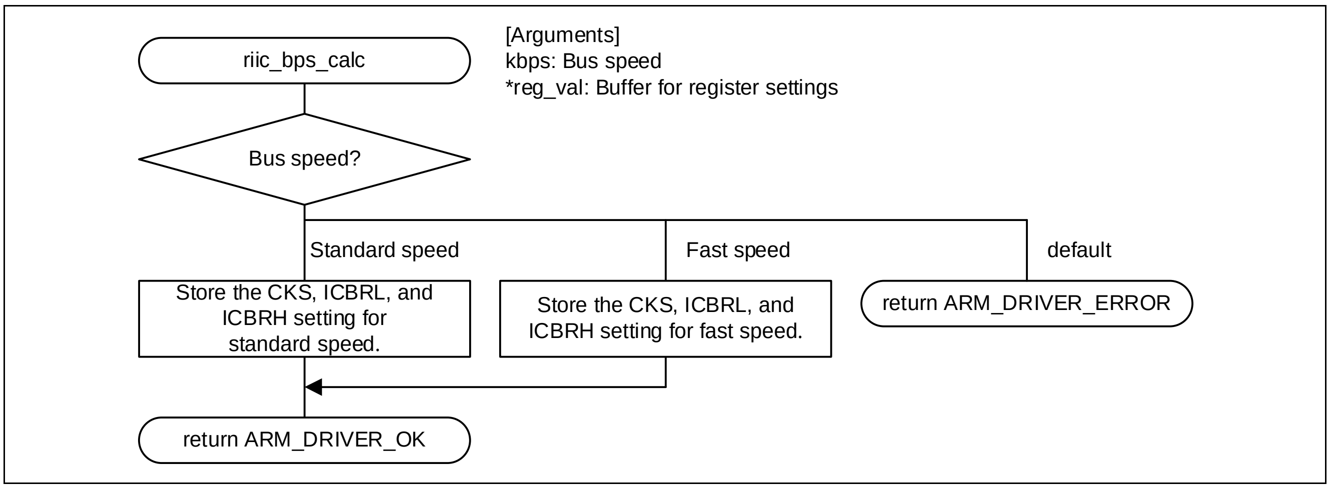

riic_bps_calc
[Arguments]
kbps: Bus speed
*reg_val: Buffer for register settings
Bus speed?
Standard speed
Fast speed
default
Store the CKS, ICBRL, and ICBRH setting for standard speed.
Store the CKS, ICBRL, and ICBRH setting for fast speed.
return ARM_DRIVER_ERROR
return ARM_DRIVER_OK

Figure 4-16 riic_bps_calc Function Processing Flow (for RIIC_BUS_SPEED_CAL_ENABLE = 0)

4.1.14 riic_setting Function

Table 4-15 riic_setting Function Specifications

Format	static int32_t riic_setting(uint16_t bps, st_i2c_resources_t *p_i2c)	
Description	Makes register settings of the RIIC.	
Argument	uint16_t bps: Bus speed	
	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C concerned.	
Return value	ARM_DRIVER_OK	Register setting succeeded
	ARM_DRIVER_ERROR	Register setting failed
Remarks	–	

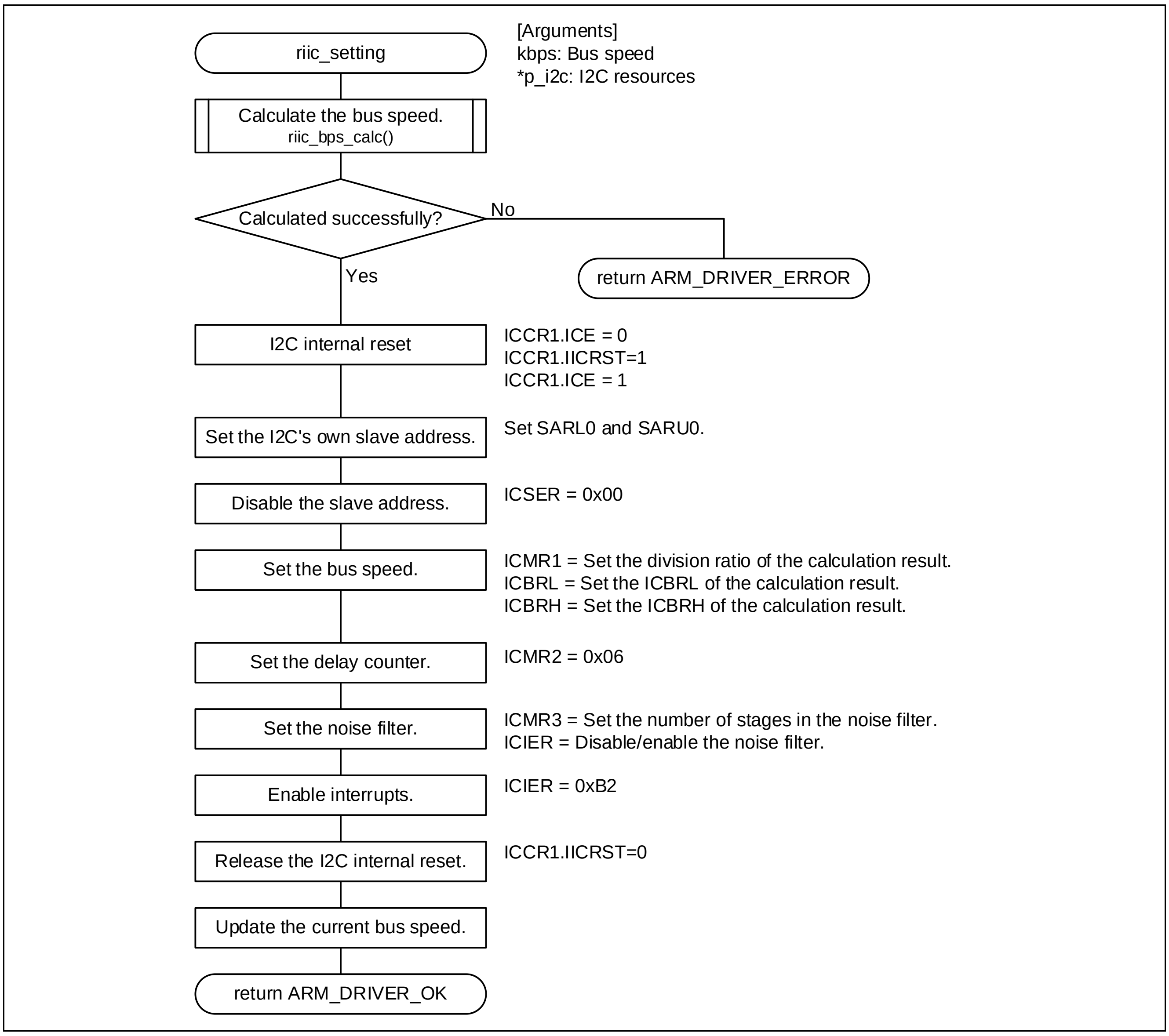


Figure 4-17 riic_setting Function Processing Flow

4.1.15 riic_reset Function

Table 4-16 riic_reset Function Specifications

Format	static void riic_reset(st_i2c_resources_t *p_i2c, bool en_recv)
Description	Performs the initialization of the RIIC.
Argument	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C concerned.
	bool en_recv RIIC_RECV_ENABLE: Performs initialization with reception enabled. RIIC_RECV_DISABLE: Performs initialization with reception disabled.
Return value	None
Remarks	–

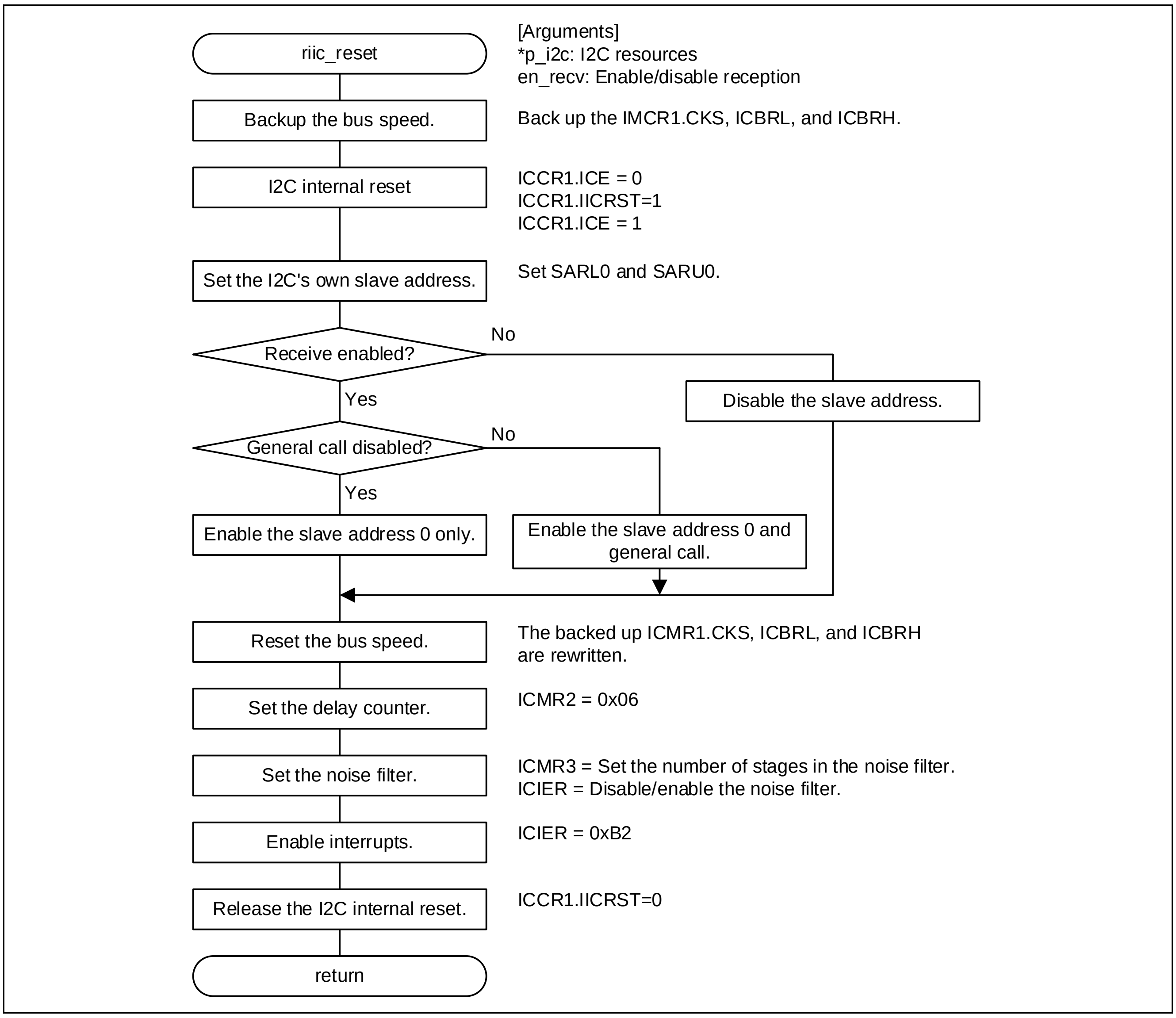
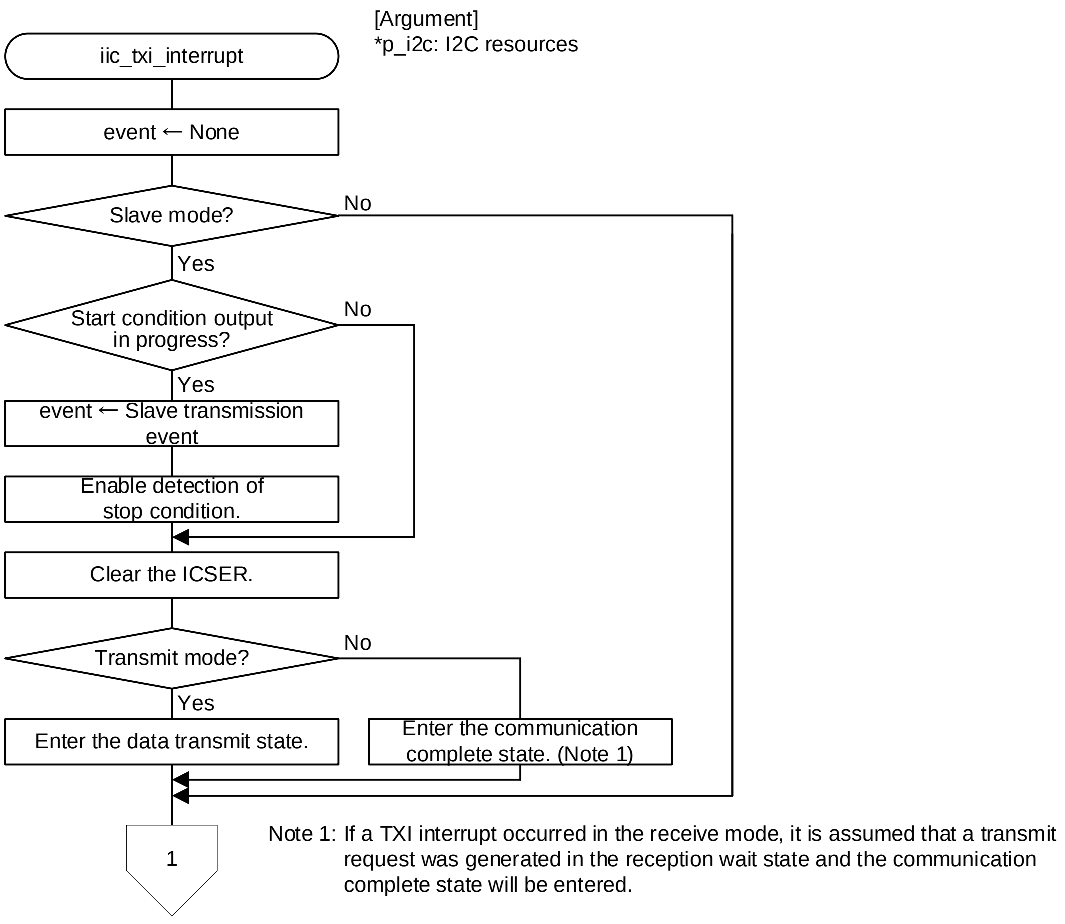

riic_reset
[Arguments]
*p_i2c: I2C resources
en_recv: Enable/disable reception
Backup the bus speed.
Back up the IMCR1.CKS, ICBRL, and ICBRH.
I2C internal reset
ICCR1.ICE = 0
ICCR1.IICRST=1
ICCR1.ICE = 1
Set the I2C's own slave address.
Set SARL0 and SARU0.
Receive enabled?
No
Yes
Disable the slave address.
General call disabled?
No
Yes
Enable the slave address 0 only.
Enable the slave address 0 and general call.
Reset the bus speed.
The backed up ICMR1.CKS, ICBRL, and ICBRH are rewritten.
Set the delay counter.
ICMR2 = 0x06
Set the noise filter.
ICMR3 = Set the number of stages in the noise filter.
ICIER = Disable/enable the noise filter.
Enable interrupts.
ICIER = 0xB2
Release the I2C internal reset.
ICCR1.IICRST=0
return

Figure 4-18 riic_reset Function Processing Flow

4.1.16 iic_txi_interrupt Function

Table 4-17 iic_txi_interrupt Function Specifications

Format	static void iic_txi_interrupt(st_i2c_resources_t *p_i2c)
Description	TXI interrupt processing
Argument	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C concerned.
Return value	None
Remarks	–


[Argument]
*p_i2c: I2C resources
iic_txi_interrupt
event ← None
Slave mode?
No
Yes
Start condition output in progress?
No
Yes
event ← Slave transmission event
Enable detection of stop condition.
Clear the ICSER.
Transmit mode?
No
Yes
Enter the data transmit state.
Enter the communication complete state. (Note 1)
1

Note 1: If a TXI interrupt occurred in the receive mode, it is assumed that a transmit request was generated in the reception wait state and the communication complete state will be entered.

Figure 4-19 iic_txi_interrupt Function Processing Flow (1/2)

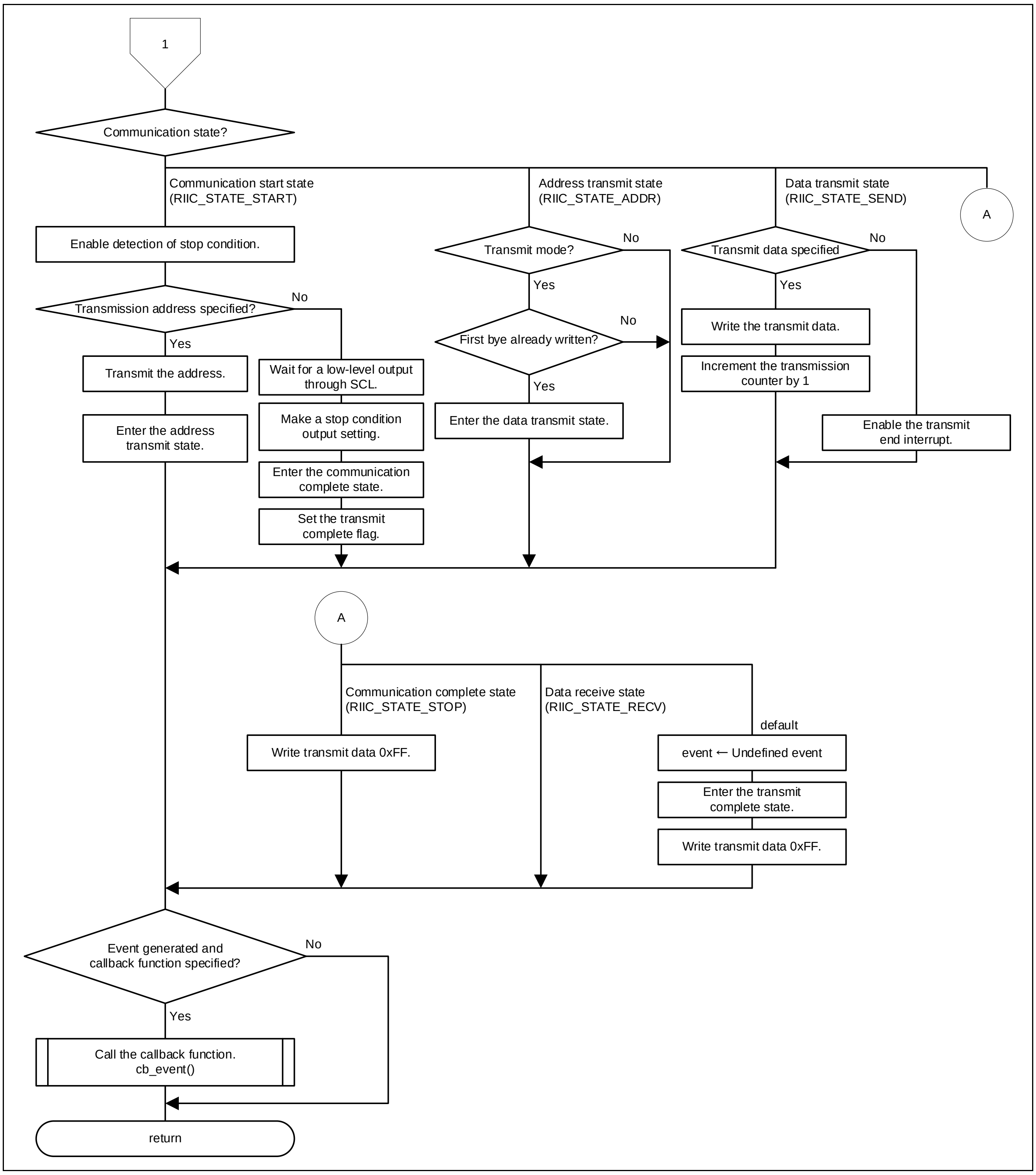

1
Communication state?
Communication start state (RIIC_STATE_START)
Address transmit state (RIIC_STATE_ADDR)
Data transmit state (RIIC_STATE_SEND)
A
Enable detection of stop condition.
Transmission address specified?
No
Yes
Transmit the address.
Enter the address transmit state.
Wait for a low-level output through SCL.
Make a stop condition output setting.
Enter the communication complete state.
Set the transmit complete flag.
Transmit mode?
No
Yes
First bye already written?
No
Yes
Enter the data transmit state.
Transmit data specified
No
Yes
Write the transmit data.
Increment the transmission counter by 1
Enable the transmit end interrupt.
A
Communication complete state (RIIC_STATE_STOP)
Data receive state (RIIC_STATE_RECV)
default
Write transmit data 0xFF.
event ← Undefined event
Enter the transmit complete state.
Write transmit data 0xFF.
Event generated and callback function specified?
No
Yes
Call the callback function. cb_event()
return

Figure 4-20 iic_txi_interrupt Function Processing Flow (2/2)

4.1.17 iic_tei_interrupt Function

Table 4-18 iic_tei_interrupt Function Specifications

Format	static void iic_tei_interrupt(st_i2c_resources_t *p_i2c)
Description	TEI interrupt processing
Argument	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C concerned.
Return value	None
Remarks	–

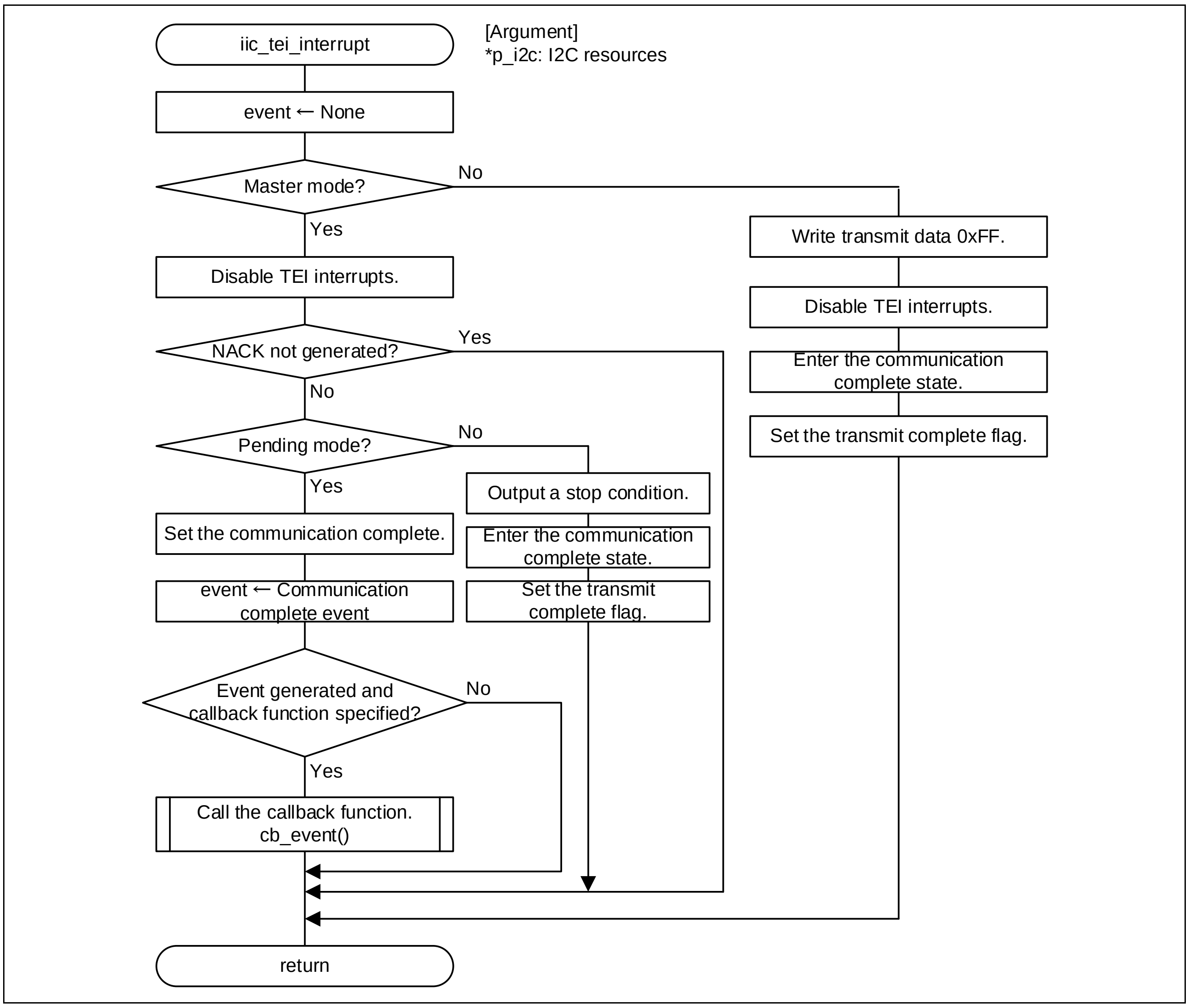

iic_tei_interrupt
[Argument]
*p_i2c: I2C resources
event ← None
Master mode?
No
Yes
Write transmit data 0xFF.
Disable TEI interrupts.
Disable TEI interrupts.
NACK not generated?
Yes
No
Enter the communication complete state.
Set the transmit complete flag.
Pending mode?
No
Yes
Output a stop condition.
Set the communication complete.
Enter the communication complete state.
event ← Communication complete event
Set the transmit complete flag.
Event generated and callback function specified?
No
Yes
Call the callback function. cb_event()
return

Figure 4-21 iic_tei_interrupt Function Processing Flow

4.1.18 iic_rxi_interrupt Function

Table 4-19 iic_rxi_interrupt Function Specifications

Format	static void iic_rxi_interrupt(st_i2c_resources_t *p_i2c)
Description	RXI interrupt processing
Argument	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C concerned.
Return value	None
Remarks	–

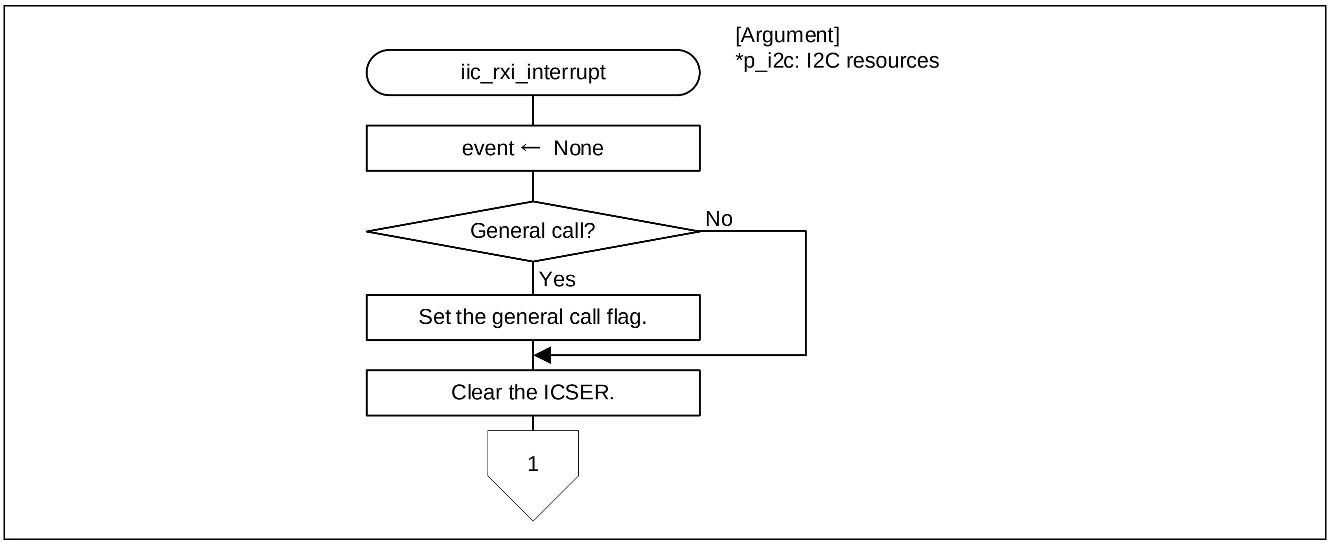


Figure 4-22 iic_rxi_interrupt Function Processing Flow (1/3)

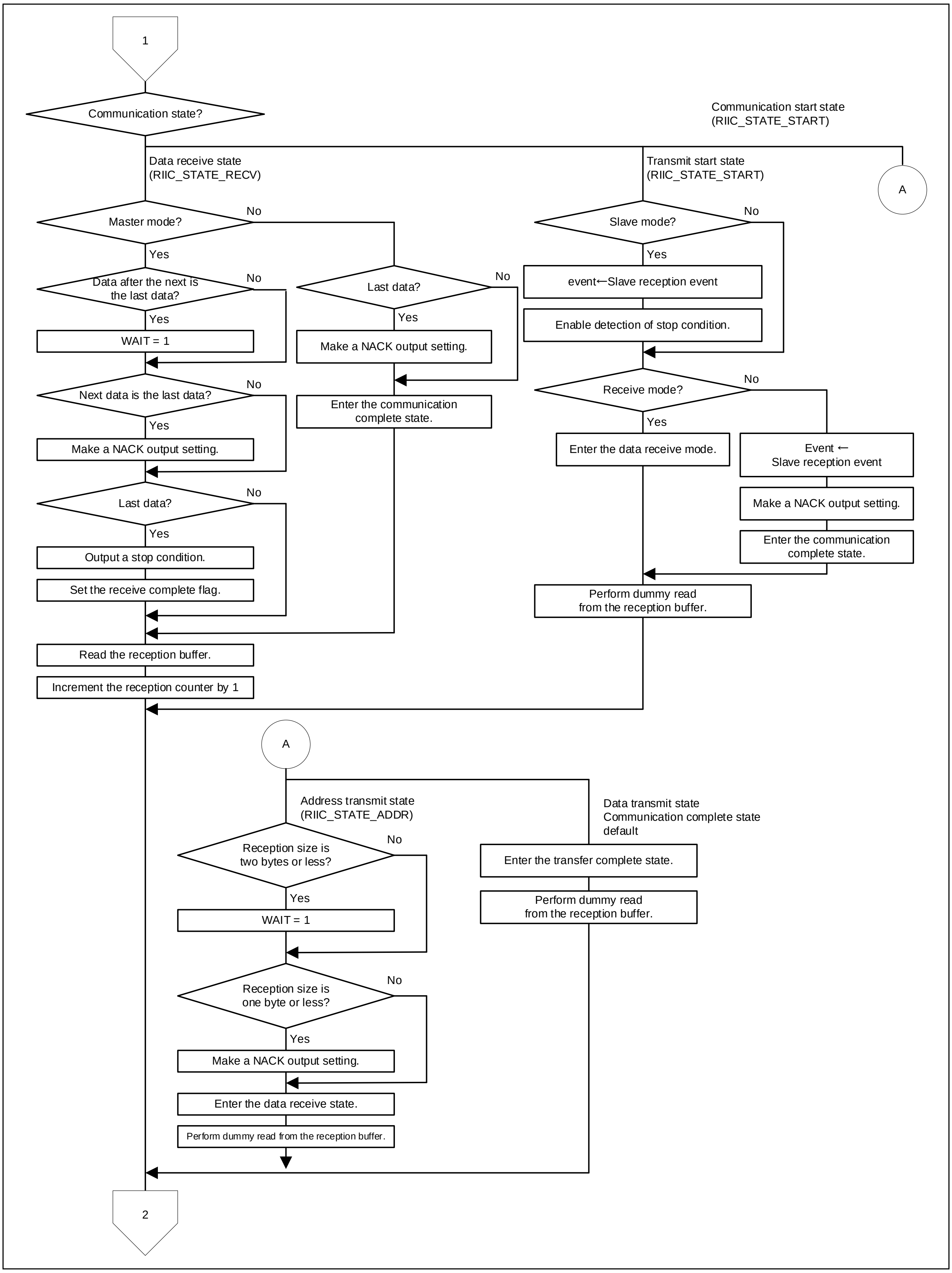

1
Communication state?
Communication start state
(RIIC_STATE_START)
Data receive state
(RIIC_STATE_RECV)
Transmit start state
(RIIC_STATE_START)
A
Master mode?
No
Yes
Data after the next is the last data?
No
Yes
WAIT = 1
Next data is the last data?
No
Yes
Make a NACK output setting.
Last data?
No
Yes
Output a stop condition.
Set the receive complete flag.
Read the reception buffer.
Increment the reception counter by 1
Last data?
No
Yes
Make a NACK output setting.
Enter the communication complete state.
Slave mode?
No
Yes
event←Slave reception event
Enable detection of stop condition.
Receive mode?
No
Yes
Enter the data receive mode.
Event ← Slave reception event
Make a NACK output setting.
Enter the communication complete state.
Perform dummy read from the reception buffer.
A
Address transmit state
(RIIC_STATE_ADDR)
Data transmit state
Communication complete state
default
Reception size is two bytes or less?
No
Yes
WAIT = 1
Reception size is one byte or less?
No
Yes
Make a NACK output setting.
Enter the data receive state.
Perform dummy read from the reception buffer.
Enter the transfer complete state.
Perform dummy read from the reception buffer.
2

Figure 4-23 iic_rxi_interrupt Function Processing Flow (2/3)

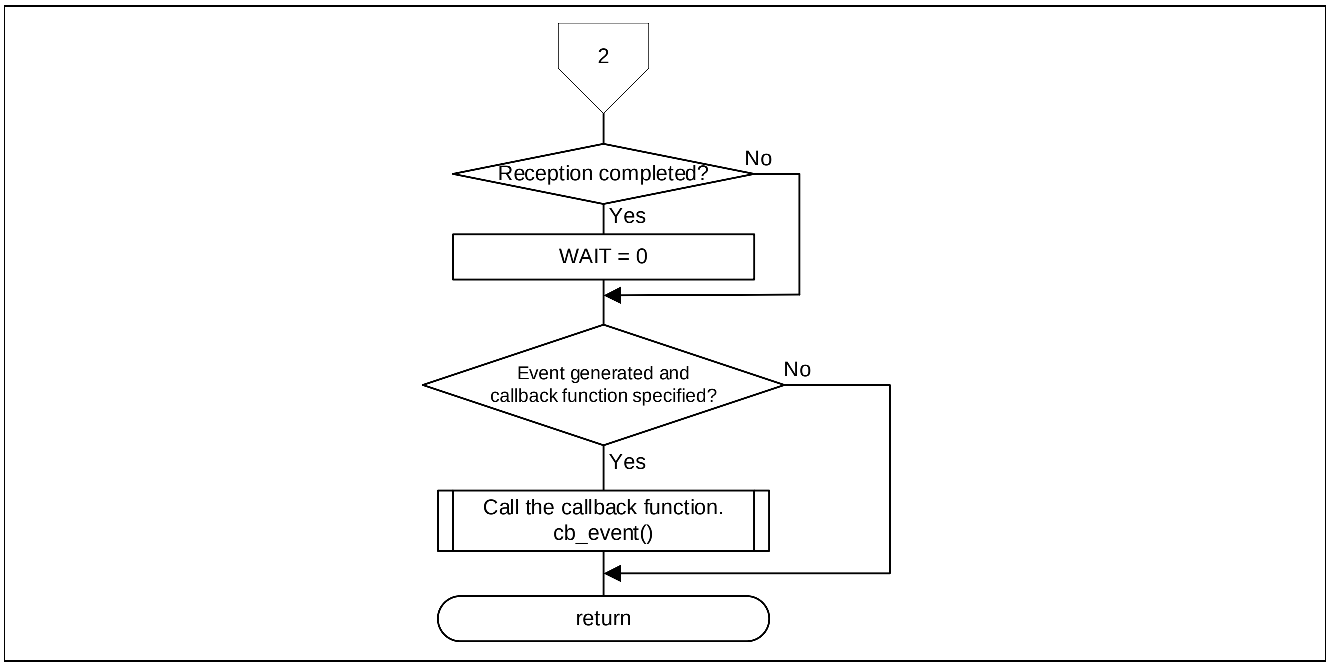

2
Reception completed?
No
Yes
WAIT = 0
Event generated and callback function specified?
No
Yes
Call the callback function. cb_event()
return

Figure 4-24 iic_rxi_interrupt Function Processing Flow (3/3)

4.1.19 iic_eei_interrupt Function

Table 4-20 iic_eei_interrupt Function Specifications

Format	static void iic_eei_interrupt(st_i2c_resources_t *p_i2c)
Description	EEI interrupt processing
Argument	st_i2c_resources_t *p_i2c: Resources of I2C Specify the resources of the I2C concerned.
Return value	None
Remarks	–

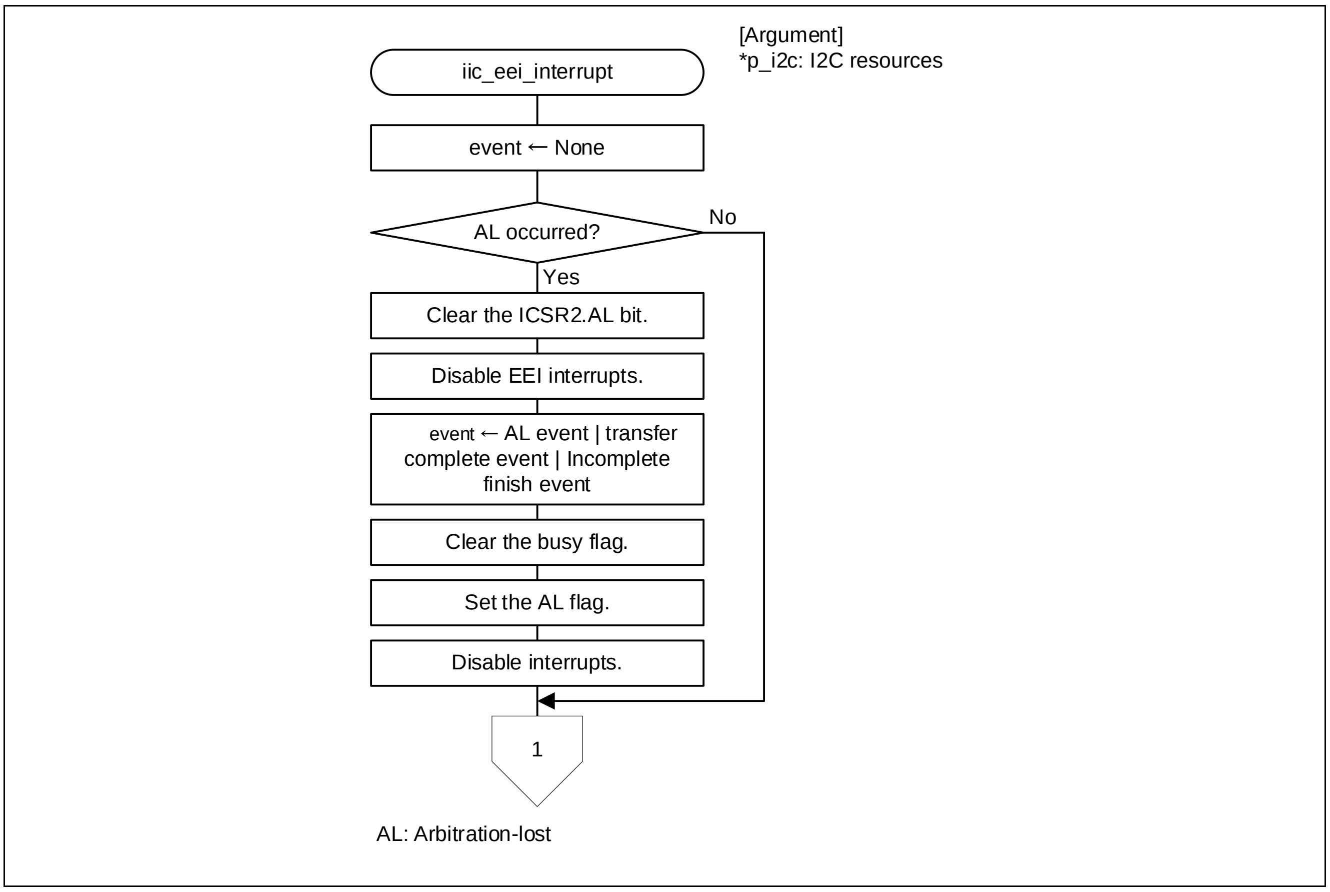


Figure 4-25 iic_eei_interrupt Function Processing Flow (1/3)

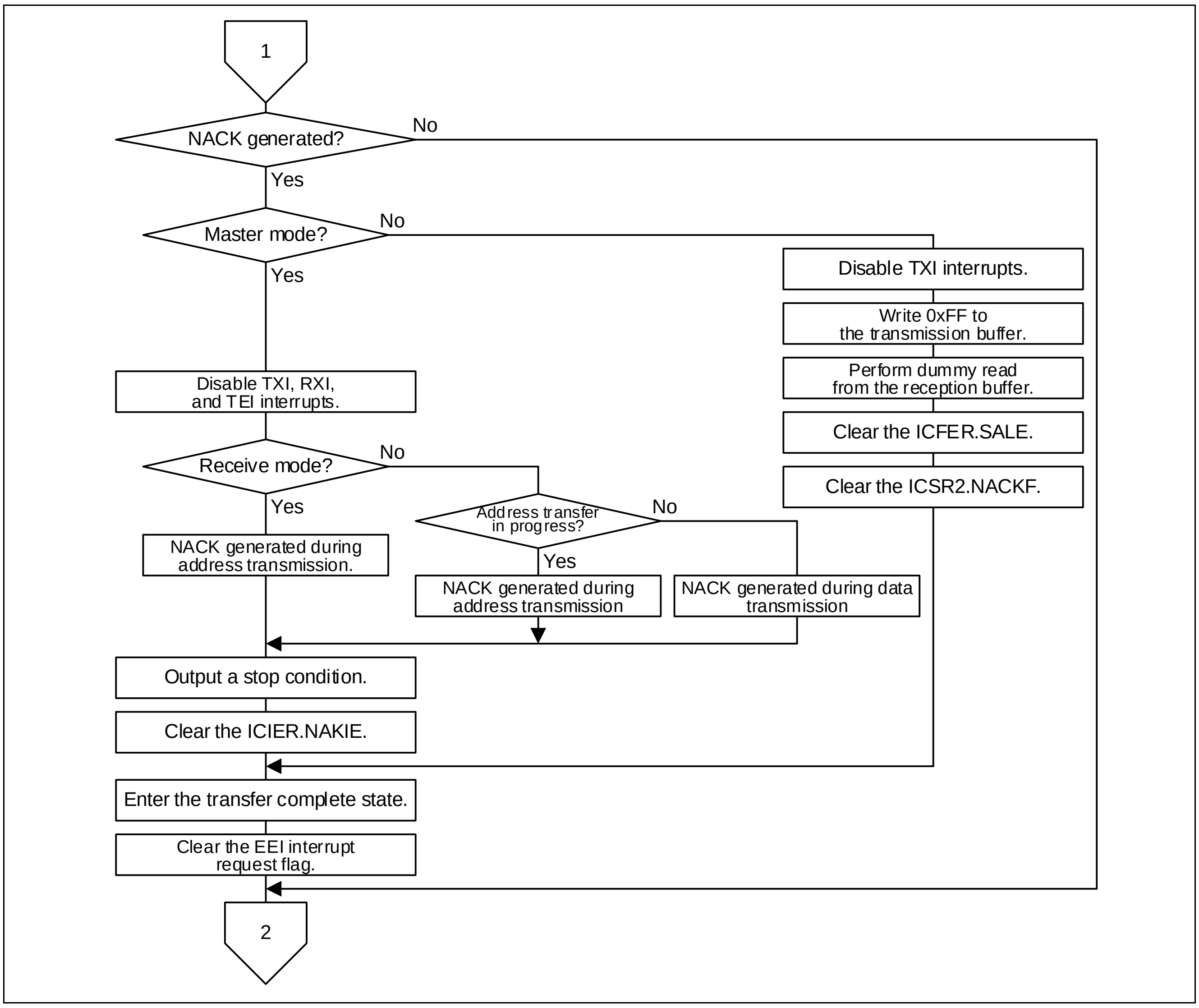

1
NACK generated?
No
Yes
Master mode?
No
Yes
Disable TXI interrupts.
Write 0xFF to the transmission buffer.
Perform dummy read from the reception buffer.
Clear the ICFER.SALE.
Clear the ICSR2.NACKF.
Disable TXI, RXI, and TEI interrupts.
Receive mode?
No
Yes
NACK generated during address transmission.
Address transfer in progress?
No
Yes
NACK generated during address transmission
NACK generated during data transmission
Output a stop condition.
Clear the ICIER.NAKIE.
Enter the transfer complete state.
Clear the EEI interrupt request flag.
2

Figure 4-26 iic_eei_interrupt Function Processing Flow (2/3)

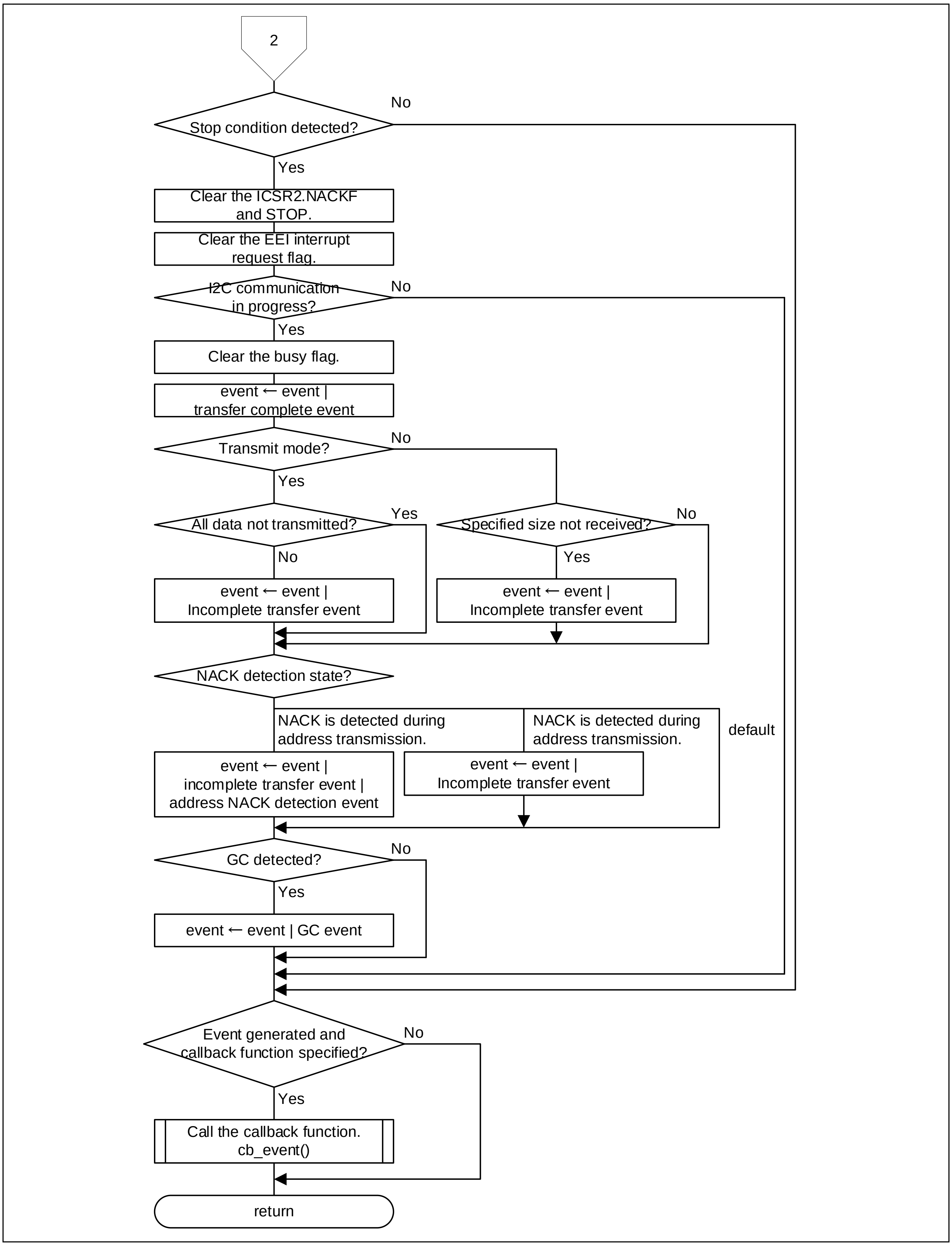

2
Stop condition detected?
No
Yes
Clear the ICSR2.NACKF and STOP.
Clear the EEI interrupt request flag.
I2C communication in progress?
No
Yes
Clear the busy flag.
event ← event | transfer complete event
Transmit mode?
No
Yes
All data not transmitted?
Yes
No
event ← event | Incomplete transfer event
Specified size not received?
No
Yes
event ← event | Incomplete transfer event
NACK detection state?
NACK is detected during address transmission.
NACK is detected during address transmission.
default
event ← event | incomplete transfer event | address NACK detection event
event ← event | Incomplete transfer event
GC detected?
No
Yes
event ← event | GC event
Event generated and callback function specified?
No
Yes
Call the callback function. cb_event()
return

Figure 4-27 iic_eei_interrupt Function Processing Flow (3/3)

4.2 Macro and Type Definitions

This section shows the macro and type definitions to be used inside the driver.

4.2.1 Macro Definition List

Table 4-21 I2C Macro Definition List

Definition	Value	Description
R_RIIC0_ENABLE	(1)	RIIC0 resource enable definition
R_RIIC1_ENABLE	(1)	RIIC1 resource enable definition
RIIC_MODE_MASTER	(1)	Master mode definition
RIIC_MODE_SLAVE	(0)	Slave mode definition
RIIC_FLAG_INITIALIZED	(1U << 0)	Initialization complete flag definition
RIIC_FLAG_POWERED	(1U << 1)	Module released flag definition
RIIC_TRANSMITTER	(0)	Definition of transmission in progress
RIIC_RECIVER	(1)	Definition of reception in progress
RIIC_MAX_DIV	((uint8_t)0x08)	Number of elements in frequency division judgment table
RIIC_STAD_SPPED_MAX	((uint16_t)100)	Standard bus speed
RIIC_FAST_SPPED_MAX	((uint16_t)400)	Fast bus speed
RIIC_ICBR_MAX	(32)	Maximum value of ICBR register
RIIC_DEFAULT_ADDR	((uint32_t)0x00000000)	Default slave address
RIIC_DEFAULT_BPS	(RIIC_STAD_SPPED_MAX)	Default bus speed
RIIC_WRITE	(0)	Write mode definition
RIIC_READ	(1)	Read mode definition
RIIC_RECV_ENABLE	(1)	Reception enable definition
RIIC_RECV_DISABLE	(0)	Reception disable definition
RIIC_ICMR3_DEF	When RIIC_NOISE_FILTER == 0 (0x00) When RIIC_NOISE_FILTER is not 0 (0x00 | RIIC_NOISE_FILTER-1)	Default definition of ICMR3 register (The value is changed depending on whether noise filter is enabled or disabled)
RIIC_ICFER_DEF	When RIIC_NOISE_FILTER == 0 (0x5A) When RIIC_NOISE_FILTER is not 0 (0x7A)	Default definition of ICFER register (The value is changed depending on whether noise filter is enabled or disabled)

4.2.2 e_i2c_driver_state_t Definition

This definition shows the status of the driver.

Table 4-22 e_i2c_driver_state_t Definition List

Definition	Value	Description
RIIC_STATE_START	0	Communication start state
RIIC_STATE_ADDR	1	Address transmit state
RIIC_STATE_SEND	2	Data transmit state
RIIC_STATE_RECV	3	Data receive state
RIIC_STATE_STOP	4	Communication complete state

4.2.3 e_i2c_nack_state_t Definition

This definition shows the status of NACK occurrence.

Table 4-23 e_i2c_nack_state_t Definition List

Definition	Value	Description
RIIC_NACK_NONE	0	NACK not generated.
RIIC_NACK_ADDR	1	NACK generated in address transfer
RIIC_NACK_DATA	2	NACK generated in data transfer

4.3 Structure Definitions

4.3.1 st_i2c_resources_t Structure

This structure composes the resources of RIIC.

Table 4-24 st_i2c_resources_t Structure

Element Name	Type	Description
*reg	volatile IIC0_Type	Shows a target RIIC register.
pin_set	r_pinset_t	Function pointer for setting pins
pin_clr	r_pinclr_t	Function pointer for releasing pins
*info	st_i2c_info_t	I2C status information
*xfer_info	st_i2c_transfer_info_t	Transfer information
lock_id	e_system_mcu_lock_t	RIIC lock ID
mstp_id	e_lpm_mstp_t	RIIC module stop ID
txi_irq	IRQn_Type	TXI interrupt number assigned in NVIC
tei_irq	IRQn_Type	TEI interrupt number assigned in NVIC
rxi_irq	IRQn_Type	RXI interrupt number assigned in NVIC
eei_irq	IRQn_Type	EEI interrupt number assigned in NVIC
txi_iesr_val	uint32_t	IESR register setting for TXI interrupt
tei_iesr_val	uint32_t	IESR register setting for TEI interrupt
rxi_iesr_val	uint32_t	IESR register setting for RXI interrupt
eei_iesr_val	uint32_t	IESR register setting for EEI interrupt
txi_priority	uint32_t	TXI interrupt priority level
tei_priority	uint32_t	TEI interrupt priority level
rxi_priority	uint32_t	RXI interrupt priority level
eei_priority	uint32_t	EEI interrupt priority level
txi_callback	system_int_cb_t	TXI interrupt callback function
tei_callback	system_int_cb_t	TEI interrupt callback function
rxi_callback	system_int_cb_t	RXI interrupt callback function
eei_callback	system_int_cb_t	EEI interrupt callback function

4.3.2 st_i2c_reg_buf_t Structure

This structure is used as buffers for temporarily storing register settings, e.g., in calculating the bus speed.

Table 4-25 st_i2c_reg_buf_t Structure

Element Name	Type	Description
cks	uint8_t	ICMR1.CKS bit setting
icbrl	uint8_t	ICBRL register setting
icbrh	uint8_t	ICBRH register setting

4.3.3 st_i2c_info_t Structure

This structure is used to manage the status of RIIC.

Table 4-26 st_i2c_info_t Structure

Element Name	Type	Description
cb_event	ARM_I2C_SignalEvent_t	Callback function to be executed when an event occurs When this value is NULL, no callback function will be executed.
status	ARM_I2C_STATUS	Status flag
flags	uint8_t	Driver setting flag Bit 0: Driver initialization status (0: Uninitialized, 1: Initialized) Bit 1: Module stop status (0: Module stop state, 1: Module stop released) Bit 2: RIIC setup status (0: Not set up, 1: Set up)
own_sla	uint32_t	I2C's own slave address
bps	uint16_t	Current bus speed
state	e_i2c_driver_state_t	Status of driver RIIC_STATE_START: RIIC start state RIIC_STATE_ADDR: Outputting address RIIC_STATE_SEND: Transmitting data RIIC_STATE_RECV: Receiving data RIIC_STATE_STOP: Communication complete
nack	e_i2c_nack_state_t	Status of NACK occurrence RIIC_NACK_NONE: NACK not generated RIIC_NACK_ADDR: NACK generated in address transfer RIIC_NACK_DATA: NACK generated in data transfer
pending	bool	Pending mode setting 0: Operates in normal mode (outputs a stop condition). 1: Operates in pending mode (does not output a stop condition).

4.3.4 st_i2c_transfer_info_t Structure

This structure is used to manage the transfer information.

Table 4-27 st_i2c_transfer_info_t Structure

Element Name	Type	Description
sla	uint32_t	Transfer destination address
rx_num	uint32_t	Reception size
tx_num	uint32_t	Transmission size
*rx_buf	uint8_t	Pointer to receive data storage buffer
*tx_buf	const uint8_t	Pointer to transmit data storage buffer
cnt	uint32_t	Current transmission or reception size
f_tx_end	uint8_t	Transmission complete flag

4.4 Calling External Functions

This section shows the external functions to be called from the I2C driver APIs.

Table 4-28 External Functions Called from I2C Driver APIs and Calling Conditions (1/2)

API	Functions Called	Conditions (Note)
Initialize	R_SYS_IrqEventLinkSet	None
	R_NVIC_SetPriority	None
	R_NVIC_GetPriority	None
	R_SYS_ResourceLock	None
	R_RIIC_Pinset_CHn (n=0,1)	None
Uninitialize	R_RIIC_Pinclr_CHn (n=0,1)	None
	R_SYS_ResourceUnlock	None
	R_NVIC_DisableIRQ	When the Uninitialize function is executed with the module stop released (Uninitialize is executed after PowerControl(ARM_POWER_FULL))
	R_LPM_ModuleStop	
PowerControl	R_NVIC_DisableIRQ	None
	R_LPM_ModuleStop	When ARM_POWER_OFF is specified (module stop state is entered) or When ARM_POWER_FULL is specified and initialization has failed
	R_LPM_ModuleStart	When ARM_POWER_FULL is specified (released from module stop state)
	R_SYS_PeripheralClockFreqGet	When ARM_POWER_FULL is specified (released from module stop state) and "1" is set in RIIC_BUS_SPEED_CAL_ENABLE (automatic bus speed calculation enabled)
MasterTransmit	R_NVIC_DisableIRQ	None
	R_SYS_IrqStatusClear	None
	R_NVIC_ClearPendingIRQ	None
	R_NVIC_EnableIRQ	None
MasterReceive	R_NVIC_DisableIRQ	None
	R_SYS_IrqStatusClear	None
	R_NVIC_ClearPendingIRQ	None
	R_NVIC_EnableIRQ	None
SlaveTransmit	R_NVIC_DisableIRQ	None
	R_SYS_IrqStatusClear	None
	R_NVIC_ClearPendingIRQ	None
	R_NVIC_EnableIRQ	None
SlaveReceive	R_NVIC_DisableIRQ	None
	R_SYS_IrqStatusClear	None
	R_NVIC_ClearPendingIRQ	None
	R_NVIC_EnableIRQ	None
GetDataCount	-	-

Note. If operation ends due to a parameter check error, the functions will not be called in some cases even when there is no condition for executing them.

Table 4-29 External Functions Called from I2C Driver APIs and Calling Conditions (2/2)

API	Functions Called	Conditions
Control	R_SYS_PeripheralClockFreqGet	When the ARM_I2C_BUS_SPEED command is executed and "1" is set in RIIC_BUS_SPEED_CAL_ENABLE (automatic bus speed calculation enabled)
	R_SYS_SoftwareDelay	When either of the following commands is executed: • ARM_I2C_BUS_CLEAR • ARM_I2C_ABORT_TRANSFER
	R_NVIC_DisableIRQ	When the ARM_I2C_ABORT_TRANSFER command is executed
	R_SYS_IrqStatusClear	
	R_NVIC_ClearPendingIRQ	
	R_NVIC_EnableIRQ	
GetStatus	-	-
GetVersion	-	-
GetCapabilities	-	-

5. Usage Notes

5.1 Registering I2C Interrupts to NVIC

Before using the I2C driver, register the receive data full interrupt (RXI), transmit end interrupt (TEI), transmit data empty interrupt (TXI), and communication error/event generation interrupt (EEI) to NVIC in r_system_cfg.h. For details, refer to "Interrupt Control" in "RE01 1500KB, 256KB Group Getting Started Guide to Development Using CMSIS Package".

If no I2C interrupt is registered in NVIC, ARM_DRIVER_ERROR will return when the ARM_I2C_Initialize function is executed.

```
• • •
#define SYSTEM_CFG_EVENT_NUMBER_ADC140_WCMPM
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 0/4/8/12/16/20/24/28 only */
#define SYSTEM_CFG_EVENT_NUMBER_IIC0_RXI
     (SYSTEM_IRQ_EVENT_NUMBER0)  /*!< Numbers 0/4/8/12/16/20/24/28 only */
#define SYSTEM_CFG_EVENT_NUMBER_CCC_PRD
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 0/4/8/12/16/20/24/28 only */
• • •
#define SYSTEM_CFG_EVENT_NUMBER_ADC140_WCMPUM
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 1/5/9/13/17/21/25/29 only */
#define SYSTEM_CFG_EVENT_NUMBER_IIC0_TXI
     (SYSTEM_IRQ_EVENT_NUMBER1)  /*!< Numbers 1/5/9/13/17/21/25/29 only */
#define SYSTEM_CFG_EVENT_NUMBER_DOC_DOPCI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 1/5/9/13/17/21/25/29 only */
• • •
#define SYSTEM_CFG_EVENT_NUMBER_ADC140_GCADI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 2/6/10/14/18/22/26/30 only */
#define SYSTEM_CFG_EVENT_NUMBER_IIC0_TEI
     (SYSTEM_IRQ_EVENT_NUMBER2)  /*!< Numbers 2/6/10/14/18/22/26/30 only */
#define SYSTEM_CFG_EVENT_NUMBER_CAC_MENDI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 2/6/10/14/18/22/26/30 only */
• • •
#define SYSTEM_CFG_EVENT_NUMBER_ACMP_CMPI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 3/7/11/15/19/23/27/31 only */
#define SYSTEM_CFG_EVENT_NUMBER_IIC0_EEI
     (SYSTEM_IRQ_EVENT_NUMBER3)  /*!< Numbers 3/7/11/15/19/23/27/31 only */
#define SYSTEM_CFG_EVENT_NUMBER_CAC_OVFI
     (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)  /*!< Numbers 3/7/11/15/19/23/27/31 only */
```

Figure 5-1 Example of registering an interrupt to NVIC in r_system_cfg.h (Using RIIC0)

5.2 Power supply open control register (VOCR) setting

Use this driver after setting the power supply open control register (VOCR).

The VOCR register prevents indefinite inputs from entering the power domain that is not supplied with power. For this reason, the VOCR register is set to shut off the input signal after reset. In this state, the input signal is not propagated inside the device. For details, refer to “Control of Undefined Value Propagation Suppression in I/O Power Supply Domains” in “RE01 1500KB, 256KB Group Getting Started Guide to Development Using CMSIS Package R01AN4660”.

5.3 Pin Configuration

The terminal used with this driver must be set in pin.c. For details on pin settings, see “2.3 Pin Configuration”.

5.4 Note When Automatic Calculation of Bus Speed Is Enabled

When automatic calculation of the bus speed is enabled (RIIC_BUS_SPEED_CAL_ENABLE = 1), values that will satisfy the I2C bus specifications are calculated from the PCLKB frequency at the time the bus speed is set, and registers are set accordingly. The rise time and fall time of the SCL line are also considered at calculation. Therefore, the bus speed must be set again when PCLKB is changed.

Table 5-1 shows the prescribed values of the SCL clock in the I2C bus specifications and the rise time and fall time of the SCL line, which are used for setting the bus speed.

Table 5-1 Prescribed Values for Setting Bus Speed

Bus Speed	Item	Value
Standard mode (100 kbps)	Prescribed low period of SCL clock	4.7 us (min.)
	Prescribed high period of SCL clock	4.0 us (min.)
	Rise time of SCL line	Defined in RIIC_STAD_SCL_UP_TIME (Initial value: 1000 ns)
	Fall time of SCL line	Defined in RIIC_STAD_SCL_DOWN_TIME (Initial value: 300 ns)
Fast mode (400 kbps)	Prescribed low period of SCL clock	1.3 us (min.)
	Prescribed high period of SCL clock	0.6 us (min.)
	Rise time of SCL line	Defined in RIIC_FAST_SCL_UP_TIME (Initial value: 300 ns)
	Fall time of SCL line	Defined in RIIC_FAST_SCL_DOWN_TIME (Initial value: 300 ns)

The calculated results satisfy the I2C bus specifications, but the error of the bus speed may increase if the PCLKB frequency is low.

Table 5-2 **Examples of Register Settings by Automatic Calculation and Error (with 2-Stage Noise Filtering)**

Bus Speed	Operating Frequency PCLKB	Register Settings			Expected Output	
		ICMR1. CKS[2:0]	ICBRH. BRH[4:0]	ICBRL. BRL[4:0]	Transfer Rate	Error
100 kbps	32 MHz	011b	12 (0Ch)	15 (0FH)	99.5 kbps	0.5%
	20 MHz	010b	16 (10h)	20 (14h)	99.0 kbps	1.0%
	8 MHz	001b	12 (0Ch)	15 (0Fh)	99.5 kbps	0.5%
	2 MHz	000b	3 (03h)	1 (01H)	120.5 kbps	20.5%
	1 MHz	000b	0 (00h)	0 (00h)	88.5 kbps	11.5%
400 kbps	32 MHz	001b	6 (06h)	17 (11h)	394.1 kbps	1.5%
	20 MHz	000b	7 (07h)	21 (15h)	400.0 kbps	0.0%
	8 MHz	000b	0 (00h)	5 (05h)	404.0 kbps	1.0%
	2 MHz	000b	0 (00h)	0 (00h)	178.6 kbps	55.4%
	1 MHz	000b	0 (00h)	0 (00h)	94.3 kbps	76.4%

Note: The transfer rate in the Expected Output column is calculated by the following formula.
tr: Rise time of SCL line, tf: Fall time of SCL line, nf: Number of stages of noise filtering
At standard speed: tr = 1000 ns, tf = 300 ns
At fast speed: tr = 300 ns, tf = 300 ns
(1) For CKS[2:0] = 000b
Transfer rate = 1/{[(BRH + 3 + nf) + (BRL + 3 + nf)]/(PCLKB) + tr + tf}
(2) For CKS[2:0] ≠ 000b
Transfer rate = 1/{[(BRH + 2 + nf) + (BRL + 2 + nf)]/ (PCLKB/Division ratio) + tr + tf}

Table 5-3 **Examples of Register Settings by Automatic Calculation and Error (without Noise Filtering)**

Bus Speed	Operating Frequency PCLKB	Register Settings			Expected Output	
		ICMR1. CKS[2:0]	ICBRH. BRH[4:0]	ICBRL. BRL[4:0]	Transfer Rate	Error
100 kbps	32 MHz	011b	14 (0Eh)	17 (11H)	99.5 kbps	0.5%
	20 MHz	010b	18 (12h)	22 (16h)	99.0 kbps	1.0%
	8 MHz	001b	14 (0Eh)	17 (11h)	99.5 kbps	0.5%
	2 MHz	000b	5 (05h)	6 (06H)	102.0 kbps	2.0%
	1 MHz	000b	1 (01h)	2 (02h)	97.1 kbps	2.9%
400 kbps	32 MHz	001b	7 (07h)	18 (12h)	414.5 kbps	3.6%
	20 MHz	000b	9 (09h)	23 (17h)	400.0 kbps	0.0%
	8 MHz	000b	2 (02h)	7 (07h)	404.0 kbps	1.0%
	2 MHz	000b	0 (00h)	0 (00h)	277.8 kbps	30.6%
	1 MHz	000b	0 (00h)	0 (00h)	151.5 kbps	62.1%

Note: The transfer rate in the Expected Output column is calculated by the following formula.
tr: Rise time of SCL line, tf: Fall time of SCL line
At standard speed: tr = 1000 ns, tf = 300 ns
At fast speed: tr = 300 ns, tf = 300 ns
(1) For CKS[2:0] = 000b
Transfer rate = 1/{[(BRH + 3) + (BRL + 3)]/ PCLKB + tr + tf}
(2) For CKS[2:0] ≠ 000b
Transfer rate = 1/{[(BRH + 2) + (BRL + 2)]/ (PCLKB/Division ratio) + tr + tf}

6. Reference Documents

User’s Manual: Hardware

RE01 1500KB Group User’s Manual: Hardware R01UH0796
RE01 256KB Group User’s Manual: Hardware R01UH0894
(The latest version can be downloaded from the Renesas Electronics website.)

RE01 Group CMSIS Package Getting Started Guide

RE01 1500KB, 256KB Group Getting Started Guide to Development Using CMSIS Package R01AN4660

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

Technical Update/Technical News

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

User’s Manual: Development Tools

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

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Oct.10.2019	—	First edition issued
1.01	Oct.28.2019	25,62 program 10,69	Changed the definition name “RIIC_NOIZE_FILTER” to “RIIC_NOISE_FILTER” Modification to comment out default pin setting of pin.c
1.02	Dec.16.2019	—	Compatible with 256KB group
1.03	Feb.19.2020	program (256KB, 1500KB)	Modified ICMR3.ACKBT setting procedure.
1.04	Nov.05.2020	—	Error correction

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.

Corporate Headquarters

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

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/.

Trademarks