RE01 1500KB, 256KB Group

CMSIS Driver R_SPI Specifications

Summary

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

Target Device

RE01 1500KB Group

RE01 256KB Group

Contents

1. Overview

This is an SPI 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_SPI driver

Peripheral functions	Description
Serial peripheral interface (SPI)	Uses SPI to achieve SPI communication (4-wire) or clock synchronous communication (3-wire)
Data transfer controller (DTC) (Note)	When DTC control is selected, DTC is used to write data to the SPI data register (SPDR) and read data from the SPDR.
DMA controller (DMAC) (Note)	When DMAC control is selected, DMAC is used to write data to the SPI data register (SPDR) and read data from the SPDR.

Note. Used only when DMAC or DTC is specified for communication control. For details, refer to "2.4 Communication Control and NVIC Interrupt Setting".

2. Driver Configuration

This chapter describes the information required for using this driver.

2.1 File Configuration

The SPI driver conforms to the CMSIS driver package specification and consists of six files: "Driver_SPI.h" in the ARM CMSIS file storage directory," r_spi_cmsis_api.c", "r_spi_cmsis_api.h", "r_spi_cfg.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 Functions of R_SPI Driver Files

File Name	Description
Driver_SPI.h	CMSIS Driver standard header file.
r_spi_cmsis_api.c	Driver source file. This provides the main function of the driver. To use the SPI driver, you must build this file.
r_spi_cmsis_api.h	Driver header file. The macro, type, and prototype declarations used in the driver are defined here.
r_spi_cfg.h	Configuration definition file. This provides the configuration definitions that can be modified by the user.
pin.c	Pin setting file. This provides pin assignment processing for the driver.
pin.h	Pin setting header file.

```
RE01Group CMSIS Package
 ├─CMSIS
 │   └─ Driver
 │        └─ Include
 │             └─ Driver_SPI.h: CMSIS Driver header file
 └─Device
     ├─ CMSIS_Driver
     │   ├─ Src
     │   │   └─r_spi_cmsis_api.c: Driver source file
     │   └─ Include
     │       └─r_spi_cmsis_api.h: Driver header file
     ├─ Config
     │   └─ r_spi_cfg_cfg.h: Configuration definition file
     ├─ pin.c: Pin setting file
     └─ pin.h: Pin setting header file
```

Figure 2-1 File Configuration of SPI Driver

2.2 Driver APIs

The SPI driver provides channel-specific instances. To use this driver, access APIs by using function pointers to each instance. The list of the SPI driver instances, examples of instance declaration, and the APIs contained in the instance are shown in Table 2-2, Figure 2-2, and Table 2-3. Examples of access to the SPI driver are shown in Figure 2-3 and Figure 2-4.

Table 2-2 List of SPI Driver Instances

Instance	Description
ARM_DRIVER_SPI Driver_SPI0	Instance for using SPI0
ARM_DRIVER_SPI Driver_SPI1	Instance for using SPI1

```
#include "R_Driver_SPI.h"

// SPI driver instance ( SPI0 )
extern ARM_DRIVER_SPI Driver_SPI0;
ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;
```

Figure 2-2 Example of SPI Driver Instance Declaration

Table 2-3 SPI Driver APIs

API	Description	Reference
Initialize	Initializes the SPI driver (initializes RAM, registers interrupts with NVIC, and releases resource lock). When the DMA is used for transmission/reception, It also initializes the DMA module.	4.1.3
Uninitialize	Releases the SPI driver (releases the pins and transitions to the module stop state) It will also cause a transition to the module stop state if the SPI is not in the state after transitioning to the module start state, then initializing registers and releasing pins. It will also release the DMA driver if the DMA is used for transmission/reception.	4.1.4
PowerControl	Releases the SPI from the module stop state or causes a transition to the state.	4.1.5
Send	Starts transmission.	4.1.6
Receive	Starts reception.	4.1.7
Transfer	Starts transmission/reception.	4.1.8
GetDataCount	Obtains transmit/receive data count.	4.1.9
Control	Executes the control commands of the SPI. For the control commands, see section 2.4.1, SPI Control Command Definitions.	4.1.10
GetStatus	Obtains the status of the SPI.	4.1.11
GetVersion	Obtains the version of the SPI driver.	4.1.1
GetCapabilities	Obtains the capabilities of the SPI driver.	4.1.2

Table 2-4 List of SPI Control Command Capability Setting Definitions

Command	Description
ARM_SPI_MODE_INACTIVE	Causes the SPI to transition to the inactive state, and configures the pins. Initializes the SPI settings.
ARM_SPI_MODE_MASTER	Initializes the SPI in master mode, and configures the pins. Should be specified in combination with an SS operation selection definition, frame format definition, data length definition, and bit order definition. Specify a baud rate for the second argument.
ARM_SPI_MODE_SLAVE	Initializes the SPI in slave mode, and configures the pins. Should be specified in combination with an SS operation selection definition, frame format definition, data length definition, and bit order definition.
ARM_SPI_SET_BUS_SPEED	Sets the transfer speed. Specify a baud rate as the second argument.
ARM_SPI_GET_BUS_SPEED	Obtains the transfer speed.
ARM_SPI_SET_DEFAULT_TX_VALUE	Sets the transmit data (default data) to be output during reception. Set the default data value as the second argument.
ARM_SPI_CONTROL_SS	Controls the SSL0 pin. Set either of the following values as the second argument. ARM_SPI_SS_INACTIVE: SSL0 pin = inactive ARM_SPI_SS_ACTIVE: SSL0 pin = active
ARM_SPI_ABORT_TRANSFER	Aborts communication.
ARM_SPI_MODE_MASTER_SIMPLEX	Disabled (Note)
ARM_SPI_MODE_SLAVE_SIMPLEX	Disabled (Note)

Note: Not supported by the SPI driver. If this definition is specified in the Control function, ARM_DRIVER_ERROR_UNSUPPORTED will be returned.

All the capabilities required for initializing the SPI in master or slave mode should be specified when setting the mode by using the Control function.

Example:

Initializes the SPI in master mode with the following settings:

SPI operation:	Slave select control output controlled by hardware
Clock polarity:	RSPCK is low in idle state
Clock phase:	Data sampling on rising edge
	Data change on falling edge
8-bit data length	
MSB first	
100 kbps	

```
spi0Drv -> Control(ARM_SPI_MODE_MASTER | ARM_SPI_SS_MASTER_HW_OUTPUT |
ARM_SPI_MSB_LSB | ARM_SPI_CPOL0_CPHA0 | ARM_SPI_DATA_BITS(8), 100000);
```

Table 2-5 to Table 2-8 list the capabilities that can be specified with the SPI, and Figure 2-3 and Figure 2-4 show the coding examples for accessing the SPI in each mode. If the capability is not specified, default value will be effective for the capability.

Table 2-5 List of SS Operation Definitions

Definition	Description
ARM_SPI_SS_MASTER_UNUSED (default)	Does not use the SSL signals during master operation.
ARM_SPI_SS_MASTER_SW	During master operation, uses slave select control through software control.
ARM_SPI_SS_MASTER_HW_OUTPUT	During master operation, outputs slave select control through hardware control.
ARM_SPI_SS_MASTER_HW_INPUT	Disabled (Note)
ARM_SPI_SS_SLAVE_HW	During slave operation, monitors slave select input through hardware control.
ARM_SPI_SS_SLAVE_SW	During slave operation, monitors slave select input through software control.

Note: Not supported by the RE01 SPI driver.

Table 2-6 List of SPI Frame Format Definitions

Definition	Description
ARM_SPI_CPOL0_CPHA0 (default)	Clock polarity (0): RSPCK is low in idle state. Clock phase (0): Data is sampled on rising edge, and changes on falling edge.
ARM_SPI_CPOL0_CPHA1	Clock polarity (0): RSPCK is low in idle state. Clock phase (1): Data changes on rising edge, and is sampled on falling edge.
ARM_SPI_CPOL1_CPHA0	Clock polarity (1): RSPCK is high in idle state. Clock phase (0): Data is sampled on rising edge, and changes on falling edge.
ARM_SPI_CPOL1_CPHA1	Clock polarity (1): RSPCK is high in idle state. Clock phase (1): Data changes on rising edge, and is sampled on falling edge.
ARM_SPI_TI_SSI	Disabled (Note)
ARM_SPI_MICROWIRE	Disabled (Note)

Note: Not supported by the RE01 SPI driver.

Table 2-7 List of Data Length Definitions

Definition	Description
ARM_SPI_DATA_BITS (n)	For n, specify the data length in bits to be used. Data length can be selected from among 8 to 16, 20, 24, and 32.

Table 2-8 List of Bit Order Definitions

Definition	Description
ARM_SPI_MSB_LSB (default)	MSB first
ARM_SPI_LSB_MSB	LSB first

```
#include "Driver_SPI.h"

static void spi_callback (uint32_t event);

// SPI driver instance ( SPI0 )
extern ARM_DRIVER_SPI Driver_SPI0;
ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;

// Receive Buffer

static uint8_t tx_data[3] = {0x01, 0x02, 0x03};
static uint8_t rx_data[3];
main()
{

    (void)spi0Drv->Initialize(spi_callback);                /* Initialize the SPI driver */
    (void)spi0Drv->PowerControl(ARM_POWER_FULL);            /* Release the SPI from module stop state
*/
    (void)spi0Drv->Control(ARM_SPI_MODE_MASTER |            /* Master mode */
                           ARM_SPI_CPOL0_CPHA0 |            /* Clock polarity: 0, clock phase: 0 */
                           ARM_SPI_LSB_MSB |                /* LSB first */
                           ARM_SPI_SS_MASTER_HW_OUTPUT |    /* Slave select control by HW */
                           ARM_SPI_DATA_BITS(8) ,           /* Data length: 8 bits */
                           100000));                        /* Transfer rate: 100kbps */
    (void)spi0Drv-> Transfer (&tx_data[0], &rx_data[0], 3); /* Start 3-byte transmission/reception */

    while(1);
}

/****************************************************************************************************
* callback function
****************************************************************************************************/
static void spi_callback(uint32_t event)
{
    switch( event )
    {
        case ARM_SPI_EVENT_TRANSFER_COMPLETE:
        {
            /* Describe the processing to be performed when communication terminates normally */
        }
        break;

        case ARM_SPI_EVENT_DATA_LOST:
        //case ARM_SPI_EVENT_MODE_FAULT:
        default:
        {
            /* Describe the processing to be performed when a communication error occurs (Note) */
        }
        break;
    }

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

Figure 2-3   Example of Access to SPI Driver (Master Mode)

Note: An ARM_SPI_EVENT_MODE_FAULT event does not occur for this driver.

```
#include "Driver_SPI.h"

static void spi_callback (uint32_t event);

// SPI driver instance ( SPI0 )
extern ARM_DRIVER_SPI Driver_SPI0;
ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;

// Receive Buffer

static uint8_t tx_data[3] = {0x01, 0x02, 0x03};
static uint8_t rx_data[3];
main()
{

    (void)spi0Drv->Initialize(spi_callback);                /* Initialize the SPI driver */
    (void)spi0Drv->PowerControl(ARM_POWER_FULL);            /* Release the SPI from module stop state
*/
    (void)spi0Drv->Control(ARM_SPI_MODE_SLAVE |             /* Slave mode */
                           ARM_SPI_CPOL0_CPHA0 |            /* Clock polarity: 0, clock phase: 0 */
                           ARM_SPI_LSB_MSB |                /* LSB first */
                           ARM_SPI_SS_SLAVE_HW |            /* Monitor the slave select input controlled
by HW */
                           ARM_SPI_DATA_BITS(8) ,           /* Data length: 8 bits */
                           100000);                         /* Transfer rate: 100kbps */
    (void)spi0Drv-> Transfer (&tx_data[0], &rx_data[0], 3)             /* Start 3-byte
transmission/reception */

    while(1);
}

/****************************************************************************************************
* callback function
****************************************************************************************************/
static void spi_callback(uint32_t event)
{
    switch( event )
    {
        case ARM_SPI_EVENT_TRANSFER_COMPLETE:
        {
            /* Describe the processing to be performed when communication terminates normally */
        }
        break;

        case ARM_SPI_EVENT_DATA_LOST:
        //case ARM_SPI_EVENT_MODE_FAULT:
        default:
        {
            /* Describe the processing to be performed when a communication error occurs (Note) */
        }
        break;
    }

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

Figure 2-4 Example of Access to SPI Driver (Slave Mode)

Note: An ARM_SPI_EVENT_MODE_FAULT event does not occur for this driver.

2.3 Pin Configuration

The pins to be used by this driver are set and released with the R_RSPI_Pinset_CHn (n=0, 1) and R_RSPI_Pinclr_CHn functions in pin.c. The R_RSPI_Pinset_CHn function is called when transmission or reception is enabled by the Control function. The R_RSPI_Pinclr_CHn function is called when transmission or reception is disabled by the Control, PowerControl or Uninitialize function.

Select the pin to be used by editing the R_RSPI_Pinset_CHn and R_RSPI_Pinclr_CHn (n = 0, 1) functions of pin.c. For details on pin settings, see “5.2 Pin Configuration”.

Pin names of SPI function have added _A, _B, _C, and _D suffixes. When assigning SPI functions, select the functional pins with the same suffix. (note)

Note. In the case of SPI function, the same signal names are present with the suffixes “_A”, “_B” “_C” and “_D” attached. These indicate groups in terms of timing adjustment, and signals from different groups cannot be used at the same time. The exceptions are the RSPCKA_C and MOSIA_C signals for the SPI and the SSLB0_D signal, which can be used at the same time as signals from group B.

2.4 Communication Control and NVIC Interrupt Setting

The SPI driver uses the interrupt handling process by default for transmission control (writing transmit data to the transmit buffer) and reception control (storing receive data in the specified buffer). Changing the set values of the transmission/reception control definition in r_spi_cfg.h allows the use of the DMAC or DTC to control transmission and reception.

Table 2-10 lists the definitions to set the transmission and reception control methods, and Table 2-11 lists the definitions of the transmission/reception control methods.

Table 2-9 Definitions to Set Transmission/Reception Control Methods (n = 0, 1)

Definition	Initial Value	Description
SPIn_TRANSMIT_CONTROL	SPI_USED_INTERRUPT	SPIn transmission control (Initial value: interrupt)
SPIn_RECEIVE_CONTROL	SPI_USED_INTERRUPT	SPIn reception control (Initial value: interrupt)

Table 2-10 Definitions of Transmission/Reception Control Methods

Definition	Value	Description
SPI_USED_INTERRUPT	(0)	Uses an interrupt for transmission/reception control.
SPI_USED_DMAC0	(1<<0)	Uses DMAC0 for transmission/reception control.
SPI_USED_DMAC1	(1<<1)	Uses DMAC1 for transmission/reception control.
SPI_USED_DMAC2	(1<<2)	Uses DMAC2 for transmission/reception control.
SPI_USED_DMAC3	(1<<3)	Uses DMAC3 for transmission/reception control.
SPI_USED_DTC	(1<<15)	Uses DTC for transmission/reception control.

It is necessary to register the interrupts used for communication control with 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 2-12 shows the definition of NVIC registration for each intended use. Figure 2-5 shows the coding example for registering the interrupts with the NVIC.

Table 2-11 Definitions of NVIC Registration for Each Intended Use (n = 0, 1, m = 0 to 3)

Mode	Intended Use	Definition of NVIC Registration
Master	Transmission only	[When interrupts or DTC is used for transmission control] SYSTEM_CFG_EVENT_NUMBER_SPIn_SPTI [When DMAC is used for transmission control] SYSTEM_CFG_EVENT_NUMBER_DMACm_INT
		SYSTEM_CFG_EVENT_NUMBER_SPIn_SPII
	Transmission and reception, or reception only (Note)	[When interrupts or DTC is used for transmission control] SYSTEM_CFG_EVENT_NUMBER_SPIn_SPTI [When DMAC is used for transmission control] SYSTEM_CFG_EVENT_NUMBER_DMACm_INT
		[When interrupts or DTC is used for reception control] SYSTEM_CFG_EVENT_NUMBER_SPIn_SPRI [When DMAC is used for reception control] None
		SYSTEM_CFG_EVENT_NUMBER_SPIn_SPII SYSTEM_CFG_EVENT_NUMBER_SPIn_SPEI
Slave	Transmission only	[When interrupts or DTC is used for transmission control] SYSTEM_CFG_EVENT_NUMBER_SPIn_SPTI [When DMAC is used for transmission control] SYSTEM_CFG_EVENT_NUMBER_DMACm_INT
		SYSTEM_CFG_EVENT_NUMBER_SPIn_SPTEND SYSTEM_CFG_EVENT_NUMBER_SPIn_SPEI
	Transmission and reception, or reception only (Note)	[When interrupts or DTC is used for transmission control] SYSTEM_CFG_EVENT_NUMBER_SPIn_SPTI [When DMAC is used for transmission control] SYSTEM_CFG_EVENT_NUMBER_DMACm_INT
		[When interrupts or DTC is used for reception control] SYSTEM_CFG_EVENT_NUMBER_SPIn_SPRI [When DMAC is used for reception control] SYSTEM_CFG_EVENT_NUMBER_DMACm_INT
		SYSTEM_CFG_EVENT_NUMBER_SPIn_SPEI

Note: Since dummy data is transmitted even when used for reception only, transmission-side settings are also necessary.

```
 . . .

#define SYSTEM_CFG_EVENT_NUMBER_SCI0_AM
            (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)   /*!< Numbers 0/4/8/12/16/20/24/28 only */
#define SYSTEM_CFG_EVENT_NUMBER_SPI0_SPRI
            (SYSTEM_IRQ_EVENT_NUMBER0)             /*!< Numbers 0/4/8/12/16/20/24/28 only */
#define SYSTEM_CFG_EVENT_NUMBER_SOL_DH
            (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)   /*!< Numbers 0/8/16/24 only */
 . . .
#define SYSTEM_CFG_EVENT_NUMBER_SCI0_TXI
           (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)    /*!< Numbers 1/5/9/13/17/21/25/29 only */
#define SYSTEM_CFG_EVENT_NUMBER_SPI0_SPTI
           (SYSTEM_IRQ_EVENT_NUMBER1)              /*!< Numbers 1/5/9/13/17/21/25/29 only */
#define SYSTEM_CFG_EVENT_NUMBER_SOL_DL
           (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)    /*!< Numbers 1/9/17/25 only */
 . . .
#define SYSTEM_CFG_EVENT_NUMBER_SCI0_TEI
           (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)    /*!< Numbers 2/6/10/14/18/22/26/30 only */
#define SYSTEM_CFG_EVENT_NUMBER_SPI0_SPII
           (SYSTEM_IRQ_EVENT_NUMBER2)            /*!< Numbers 2/6/10/14/18/22/26/30 only */
#define SYSTEM_CFG_EVENT_NUMBER_SPI0_SPTEND
           (SYSTEM_IRQ_EVENT_NUMBER6)            /*!< Numbers 2/6/10/14/18/22/26/30 only */
#define SYSTEM_CFG_EVENT_NUMBER_USBFS_USBI
           (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED)    /*!< Numbers 2/10/18/26 only */
 . . .
```

Figure 2-5 Example of registering an interrupt to NVIC in r_system_cfg.h (SPI0 used)

2.5 Macro and Type Definitions

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

2.5.1 SPI Control Command Definitions

The SPI control commands contain the SPI mode and capability definitions used as the first arguments of the Control function.

Each control command is a combination of the capability setting definition, and/or data length setting definition, SS operation selecting definition, bit order setting definition, and frame format setting definition. When using capability setting to set the SPI communication mode, also set the data length, SS operation, bit order, and frame format.

Figure 2-6 shows the structure of the SPI control command containing various definitions, and Table 2-13 to Table 2-17 show the setting definitions of each capability.

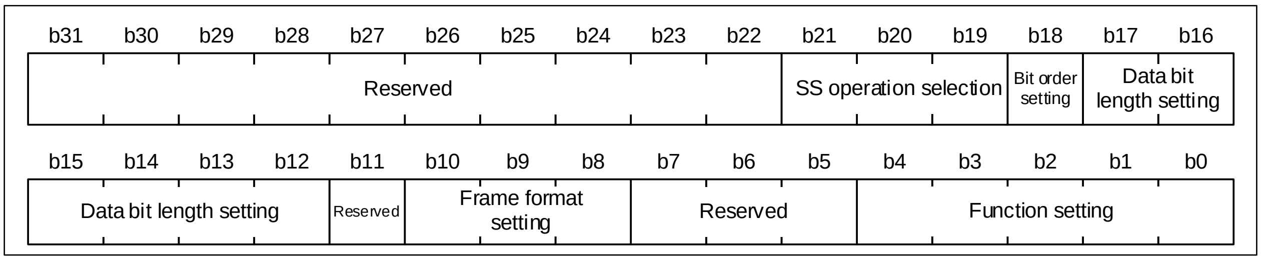


Figure 2-6 Structure of SPI Control Command Containing Definitions

Table 2-12 SPI Control Command Capability Setting Definitions

Definition	Value	Description
ARM_SPI_MODE_INACTIVE	(0x00UL << ARM_SPI_CONTROL_Pos)	Sets SPI inactive mode.
ARM_SPI_MODE_MASTER	(0x01UL << ARM_SPI_CONTROL_Pos)	Sets SPI master mode.
ARM_SPI_MODE_SLAVE	(0x02UL << ARM_SPI_CONTROL_Pos)	Sets SPI slave mode.
ARM_SPI_SET_BUS_SPEED	(0x10UL << ARM_SPI_CONTROL_Pos)	Sets bus speed.
ARM_SPI_GET_BUS_SPEED	(0x11UL << ARM_SPI_CONTROL_Pos)	Obtains bus speed.
ARM_SPI_SET_DEFAULT_TX_VALUE	(0x12UL << ARM_SPI_CONTROL_Pos)	Sets default data.
ARM_SPI_CONTROL_SS	(0x13UL << ARM_SPI_CONTROL_Pos)	Sets SSL0 pin control.
ARM_SPI_ABORT_TRANSFER	(0x14UL << ARM_SPI_CONTROL_Pos)	Aborts communication.
ARM_SPI_MODE_MASTER_SIMPLEX	(0x03UL << ARM_SPI_CONTROL_Pos)	Disabled (Note)
ARM_SPI_MODE_SLAVE_SIMPLEX	(0x04UL << ARM_SPI_CONTROL_Pos)	Disabled (Note)

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

Table 2-13 SPI Control Command SS Operation Selecting Definitions

Definition	Value	Description
ARM_SPI_SS_MASTER_UNUSED	(0UL << ARM_SPI_SS_MASTER_MODE_Pos)	Does not use the SSL signals during master operation.
ARM_SPI_SS_MASTER_SW	(1UL << ARM_SPI_SS_MASTER_MODE_Pos)	Uses slave select control through software control during master operation.
ARM_SPI_SS_MASTER_HW_OUTPUT	(2UL << ARM_SPI_SS_MASTER_MODE_Pos)	Outputs slave select control through HW control during master operation,
ARM_SPI_SS_MASTER_HW_INPUT	(3UL << ARM_SPI_SS_MASTER_MODE_Pos)	Disabled (Note)
ARM_SPI_SS_SLAVE_HW	(0UL << ARM_SPI_SS_SLAVE_MODE_Pos)	Monitors slave select input through HW control.
ARM_SPI_SS_SLAVE_SW	(1UL << ARM_SPI_SS_SLAVE_MODE_Pos)	Monitors slave select input through software control during slave operation.

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

Table 2-14 SPI Control Command Frame Format Setting Definitions

Definition	Value	Description
ARM_SPI_CPOL0_CPHA0	(0UL << ARM_SPI_FRAME_FORMAT_Pos)	Clock polarity: 0 Clock phase: 0
ARM_SPI_CPOL0_CPHA1	(1UL << ARM_SPI_FRAME_FORMAT_Pos)	Clock polarity: 0 Clock phase: 1
ARM_SPI_CPOL1_CPHA0	(2UL << ARM_SPI_FRAME_FORMAT_Pos)	Clock polarity: 1 Clock phase: 0
ARM_SPI_CPOL1_CPHA1	(3UL << ARM_SPI_FRAME_FORMAT_Pos)	Clock polarity: 1 Clock phase: 1
ARM_SPI_TI_SSI	(4UL << ARM_SPI_FRAME_FORMAT_Pos)	Disabled (Note)
ARM_SPI_MICROWIRE	(5UL << ARM_SPI_FRAME_FORMAT_Pos)	Disabled (Note)

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

Table 2-15 SPI Control Command Data Length Setting Definition

Definition	Value	Description
ARM_SPI_DATA_BITS (n)	(((n) & 0x3F) << ARM_SPI_DATA_BITS_Pos)	Sets data length in bits. (n = 8 to 16, 20, 24 or 32)

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

Table 2-16 SPI Control Command Bit Order Setting Definitions

Definition	Value	Description
ARM_SPI_MSB_LSB	(0UL << ARM_SPI_BIT_ORDER_Pos)	MSB first
ARM_SPI_LSB_MSB	(1UL << ARM_SPI_BIT_ORDER_Pos)	LSB first

2.5.2 SPI-Specific Error Code Definitions

These are the error code definitions specific to the SPI.

Table 2-17 List of SPI-Specific Error Code Definitions

Definition	Value	Description
ARM_SPI_ERROR_MODE	(ARM_DRIVER_ERROR_SPECIFIC - 1)	The specified mode is not supported.
ARM_SPI_ERROR_FRAME_FORMAT	(ARM_DRIVER_ERROR_SPECIFIC - 2)	The specified frame format is not supported.
ARM_SPI_ERROR_DATA_BITS	(ARM_DRIVER_ERROR_SPECIFIC - 3)	The specified data length is not supported.
ARM_SPI_ERROR_BIT_ORDER	(ARM_DRIVER_ERROR_SPECIFIC - 4)	The specified bit order is not supported.
ARM_SPI_ERROR_SS_MODE	(ARM_DRIVER_ERROR_SPECIFIC - 5)	The specified slave select mode is not supported.

2.5.3 SSL Signal Definitions

These are the SSL signal control definitions used by the ARM_SPI_CONTROL_SS command of the Control function.

Table 2-18 List of Modem Control Definitions

Definition	Value	Description
ARM_SPI_SS_INACTIVE	(0)	Deactivates the SSL0 output ("H").
ARM_SPI_SS_ACTIVE	(1)	Activates the SSL0 output ("L").

2.5.4 SPI Event Code Definitions

These are the definitions of the events to be notified by callback functions.

Table 2-19 List of SPI Event Codes

Definition	Value	Description
ARM_SPI_EVENT_TRANSFER_COMPLETE	(1UL << 0)	Communication was completed.
ARM_SPI_EVENT_DATA_LOST	(1UL << 1)	Transmit/receive data was lost due to overrun error or underrun error.
ARM_SPI_EVENT_MODE_FAULT (Note)	(1UL << 2)	Mode fault error occurred.

Note: This definition is not supported by this driver.

2.6 Structure Definitions

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

2.6.1 ARM_SPI_STATUS Structure

This structure is used when the GetStatus function returns the SPI status.

Table 2-20 ARM_SPI_STATUS Structure

Element Name	Type	Description
busy	uint32_t:1	Shows communication status. 0: Waiting for communication 1: Communication in progress (busy)
data_lost	uint32_t:1	Shows overrun/underrun error status. 0: No overrun/underrun error has occurred. 1: An overrun/underrun error has occurred.
mode_fault	uint32_t:1	Unused (fixed at 0)
reserved	uint32_t:29	Reserved area

2.6.2 ARM_SPI_CAPABILITIES Structure

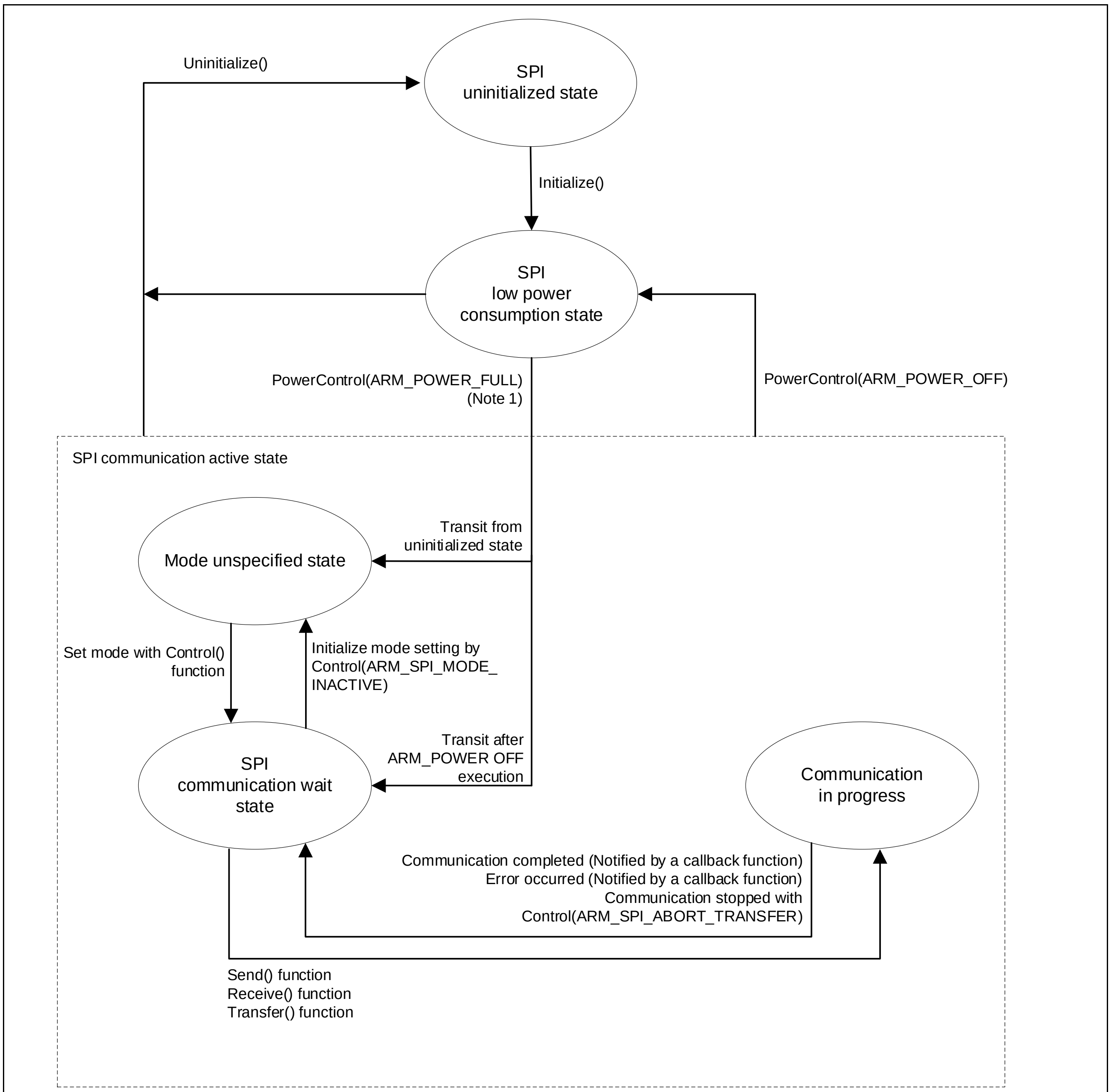
This structure is used when the GetCapabilities function returns the capabilities of the SPI.

Table 2-21 ARM_SPI_CAPABILITIES Structure

Element Name	Type	Description	Value
simplex	uint32_t:1	Enables or disables simplex mode (master/slave).	0 (disabled)
ti_ssi	uint32_t:1	Enables or disables TI synchronous serial interface.	0 (disabled)
microwire	uint32_t:1	Enables or disables Microwire interface.	0 (disabled)
event_mode_fault	uint32_t:1	Signal mode fault event: Enables or disables ARM_SPI_EVENT_MODE_FAULT.	0 (disabled)
reserved	uint32_t:29	Reserved area	-

2.7 State Transitions

The state transition diagram of the SPI driver is shown in Figure 2-7, and state-specific events and actions are shown in Table 2-22.


Uninitialize()
SPI uninitialized state
Initialize()
SPI low power consumption state
PowerControl(ARM_POWER_FULL) (Note 1)
PowerControl(ARM_POWER_OFF)
SPI communication active state
Mode unspecified state
Transit from uninitialized state
Set mode with Control() function
Initialize mode setting by Control(ARM_SPI_MODE_ INACTIVE)
Transit after ARM_POWER OFF execution
SPI communication wait state
Communication in progress
Communication completed (Notified by a callback function)
Error occurred (Notified by a callback function)
Communication stopped with Control(ARM_SPI_ABORT_TRANSFER)
Send() function
Receive() function
Transfer() function

Note 1. If a transition is made to SPI low power consumption state by PowerControl(ARM_POWER_OFF), the SPI mode is maintained. To change SPI operation mode, once enter the SPI uninitialized state and then execute initialization again.

Figure 2-7 State Transitions of SPI Driver

Table 2-22 Events and Actions Specific to SPI Driver State (Note 1)

State	Overview	Event	Action
SPI uninitialized state	State of the SPI driver after reset clear	Executing the Initialize() function	Enters the SPI low power consumption state.
SPI low power consumption state	State in which no clock is supplied to the SPI module	Executing the Uninitialize() function	Enters the SPI uninitialized state.
		Executing the PowerControl(ARM_POWER_FULL) function	Enters the mode unspecified state or the SPI communication wait state. (Note 2)
Mode unspecified state	State in which SPI mode is unspecified	Executing the Control(ARM_SPI_MODE_XXX) function (Note 3)	Enters the SPI communication wait state.
		Executing the Uninitialize() function	Enters the SPI uninitialized state.
		Executing the PowerControl(ARM_POWER_OFF) function	Enters the SPI low power consumption state.
SPI communication wait state	State in which the SPI is waiting for communication	Executing the Uninitialize() function	Enters the SPI uninitialized state.
		Executing the PowerControl(ARM_POWER_OFF) function	Enters the SPI low power consumption state.
		Executing the Send function	Enters the communicating state (Starts transmission).
		Executing the Receive function	Enters the communicating state (Starts reception).
		Executing the Transfer function	Enters the communicating state (Starts transmission/reception).
		Executing the Control(ARM_SPI_CONTROL_SS) function	Controls the SSL pin. (Note 4)
		Executing the Control(ARM_SPI_MODE_INACTIVE) function	Enters the mode unspecified state.
Communicating state	State in which SPI communication is in progress	Executing the Uninitialize() function	Enters the SPI uninitialized state.
		Executing the PowerControl(ARM_POWER_OFF) function	Enters the SPI low power consumption state.
		Terminating communication	Enters the SPI communication enabled state and calls the callback function. (Note 5)
		Error occurrence	Enters the SPI communication enabled state and calls the callback function. (Note 5)
		Disabling transmission/reception with Control() function	Enters the SPI communication wait state.
		Executing the Control(ARM_SPI_ABORT_TRANSFER) function	Aborts communication and enters the SPI communication enabled state.

Note 1. The GetVersion, GetCapabilities, and GetDataCount functions can be executed in any state.
Note 2. Enters the mode unspecified state from the SPI uninitialized state if the SPI mode has not been set.
Note 3. XXX indicates either of the following:
MASTER: master mode
SLAVE: slave mode
Note 4. Effective only if the SSL pin has been set in r_spi_cfg.h and the software slave selection control has been set to be used when the mode has been set.
Note 5. 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 SPI driver provides the SPI operation (3-wire) and clock synchronous operation (4-wire).

In SPI mode (4-wire) master mode, SSL signal output is controlled by hardware, and in slave mode, SSL signal input is monitored by hardware. For clock synchronous operation (3-wire), the SSL signal is controlled and monitored by software.

This chapter describes the procedures for initializing the SPI driver in each mode.

3.1 Master Mode

3.1.1 Initial Setting Procedure for Master Mode

The initial setting procedure for master mode is shown in Figure 3-1.

When enabling transmission and reception, register the interrupts to use with NVIC in r_system_cfg.h. For details, see section 2.4, Communication Control and NVIC Interrupt Setting.

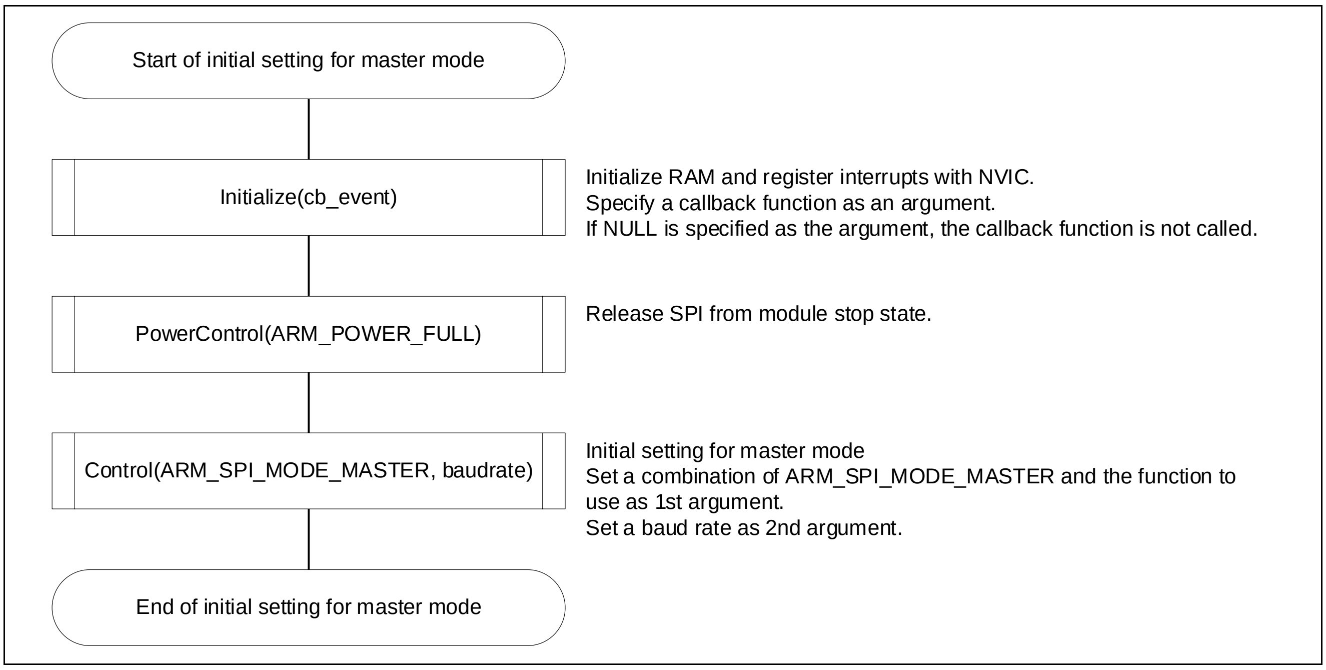

Start of initial setting for master mode
Initialize(cb_event)
Initialize RAM and register interrupts with NVIC.
Specify a callback function as an argument.
If NULL is specified as the argument, the callback function is not called.
PowerControl(ARM_POWER_FULL)
Release SPI from module stop state.
Control(ARM_SPI_MODE_MASTER, baudrate)
Initial setting for master mode
Set a combination of ARM_SPI_MODE_MASTER and the function to use as 1st argument.
Set a baud rate as 2nd argument.
End of initial setting for master mode

Figure 3-1 Master Mode Initialization Procedure

3.1.2 Transmission in Master Mode

The transmission procedure in master mode is shown in Figure 3-2.

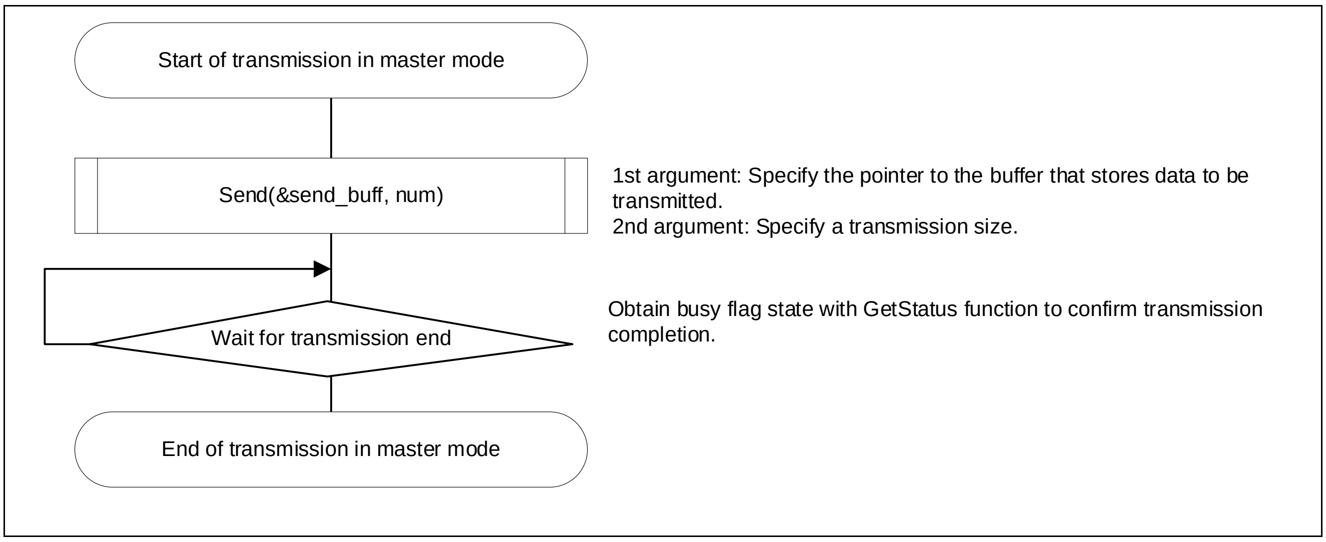
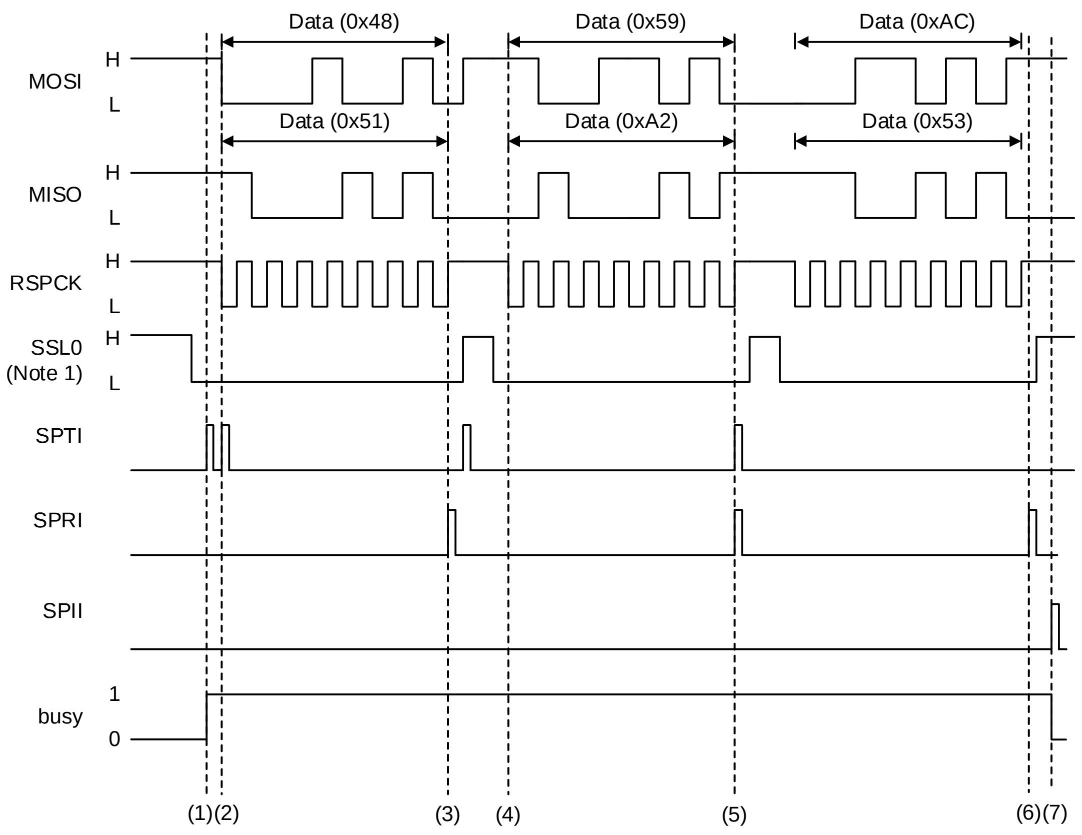

Start of transmission in master mode
Send(&send_buff, num)
1st argument: Specify the pointer to the buffer that stores data to be transmitted.
2nd argument: Specify a transmission size.
Wait for transmission end
Obtain busy flag state with GetStatus function to confirm transmission completion.
End of transmission in master mode

Figure 3-2 Transmission Procedure in Master Mode

If a callback function has been set, it is called using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument when transmission is completed.

The specific transmit operation in master mode is different depending on what method is used for communication control: the interrupt, DMAC, or DTC. Figure 3-3 shows the transmit operation when the interrupt is used for communication control. Figure 3-4 and Figure 3-5 respectively show the operations when the DMAC and the DTC are used for communication control.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-3 Transmit Operation Using Interrupt for Control (3 bytes transmitted)

(1) When the Send function is executed, the busy flag is set to “1” (communicating state).The SPTI interrupt is also generated and the first byte of data is written to the transmit data register (SPDR).

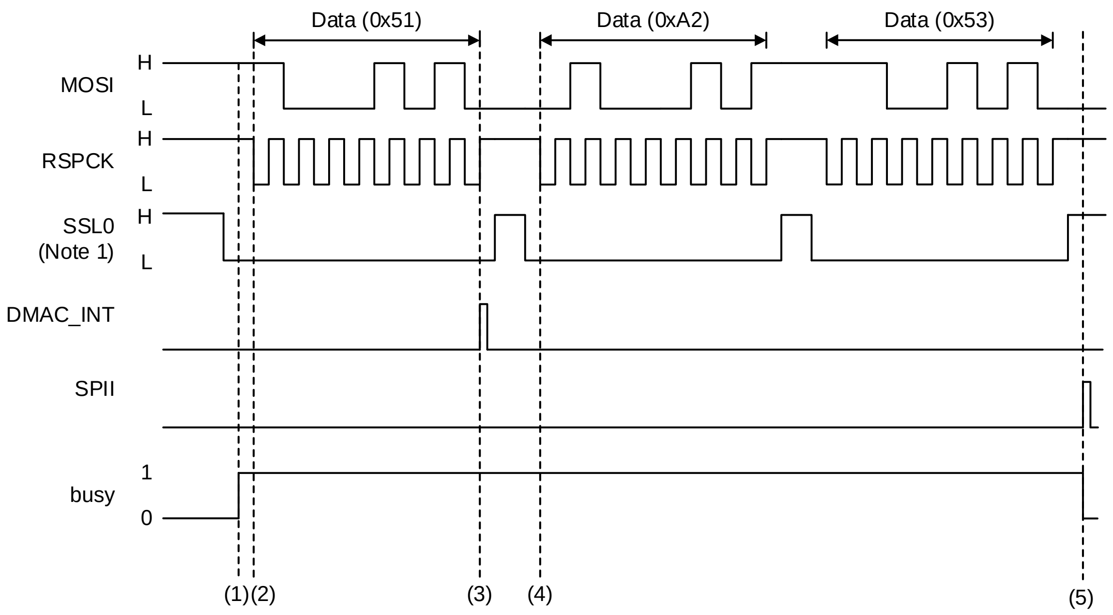
(2) At the second SPTI interrupt, the second byte of the transmit data is written to the SPDR register.

(3) At the third SPTI interrupt, the last transmit data is written to SPDR.

(4) The second byte of the data written in step 2 is output.

(5) At the SPTI interrupt after the last data written in step 3 was output, the SPTI interrupt is disabled, and the SPII interrupt is enabled.

(6) When transmission of all data is completed, the SPII interrupt occurs and the busy flag is cleared to “0” (communication wait state). In the SPII interrupt handling process, all the interrupts used for SPI control are disabled. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-4 Transmit Operation Using DMAC for Control (3 bytes transmitted)

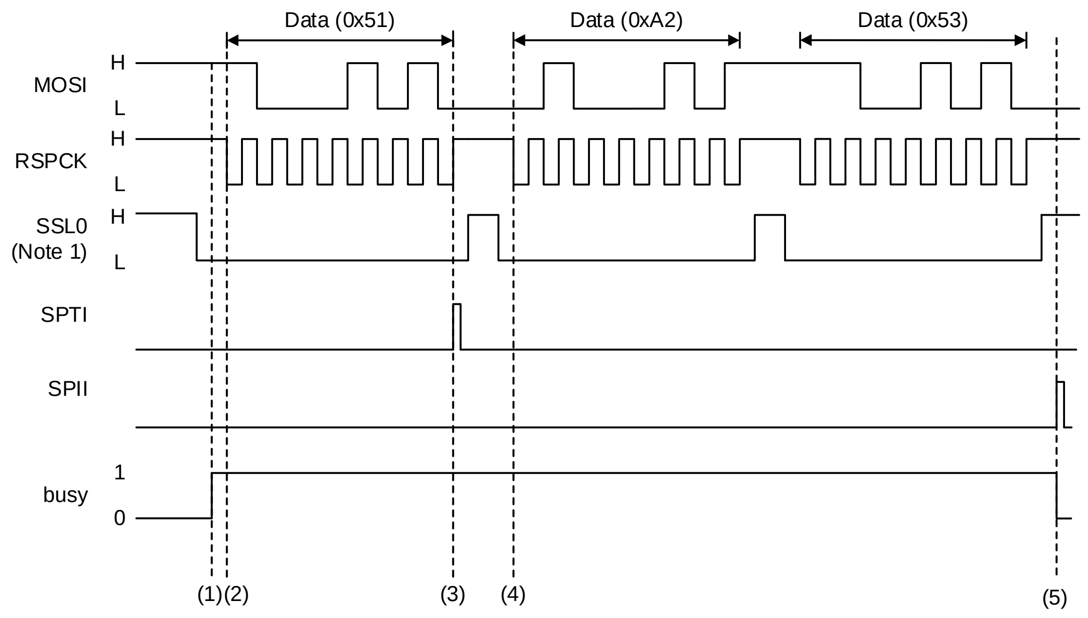
(1) When the Send function is executed, the busy flag is set to "1" (communicating state), and the SPTI interrupt is set as a DMAC transfer factor. The first byte of transmit data is DMA-transferred to the SPDR register.

(2) The second byte of the transmit data is DMA-transferred to the SPDR register.

(3) The last data is DMA-transferred. A DMAC transfer end interrupt occurs, disabling the DMAC transfer end interrupt and enabling the SPII interrupt.

(4) After the second byte of the data is output, the last data written in step 3 is output.

(5) When transmission of all data is completed, the SPII interrupt occurs and the busy flag is cleared to “0” (communication wait state). In the SPII interrupt handling process, all the interrupts used in SPI control are disabled. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-5 Transmit Operation Using DTC for Control (3 bytes transmitted)

(1) When the Send function is executed, the busy flag is set to "1" (communicating state) and the SPTI interrupt is set as a DTC transfer factor. The first byte of transmit data is DMA-transferred to the SPDR register.

(2) The second byte of the transmit data is DMA-transferred to the SPDR register.

(3) The last data is DMA-transferred. The SPTI interrupt occurs, disabling the SPTI interrupt and enabling the SPII interrupt.

(4) After the second byte of the data is output, the last data written in step 3 is output.

(5) When transmission of all data is completed, the SPII interrupt occurs and the busy flag is cleared to "0" (communication wait state). In the SPII interrupt handling process, all the interrupts used for SPI control are disabled. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument.

3.1.3 Reception in Master Mode

The reception procedure in master mode is shown in Figure 3-6.

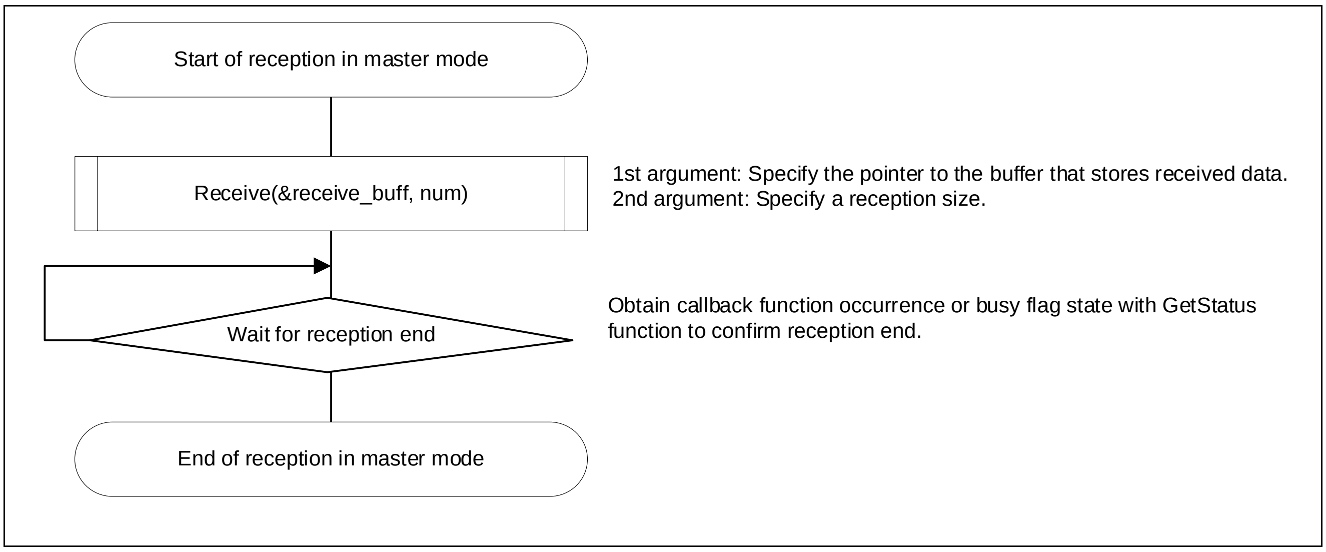

Start of reception in master mode
Receive(&receive_buff, num)
1st argument: Specify the pointer to the buffer that stores received data.
2nd argument: Specify a reception size.
Wait for reception end
Obtain callback function occurrence or busy flag state with GetStatus function to confirm reception end.
End of reception in master mode

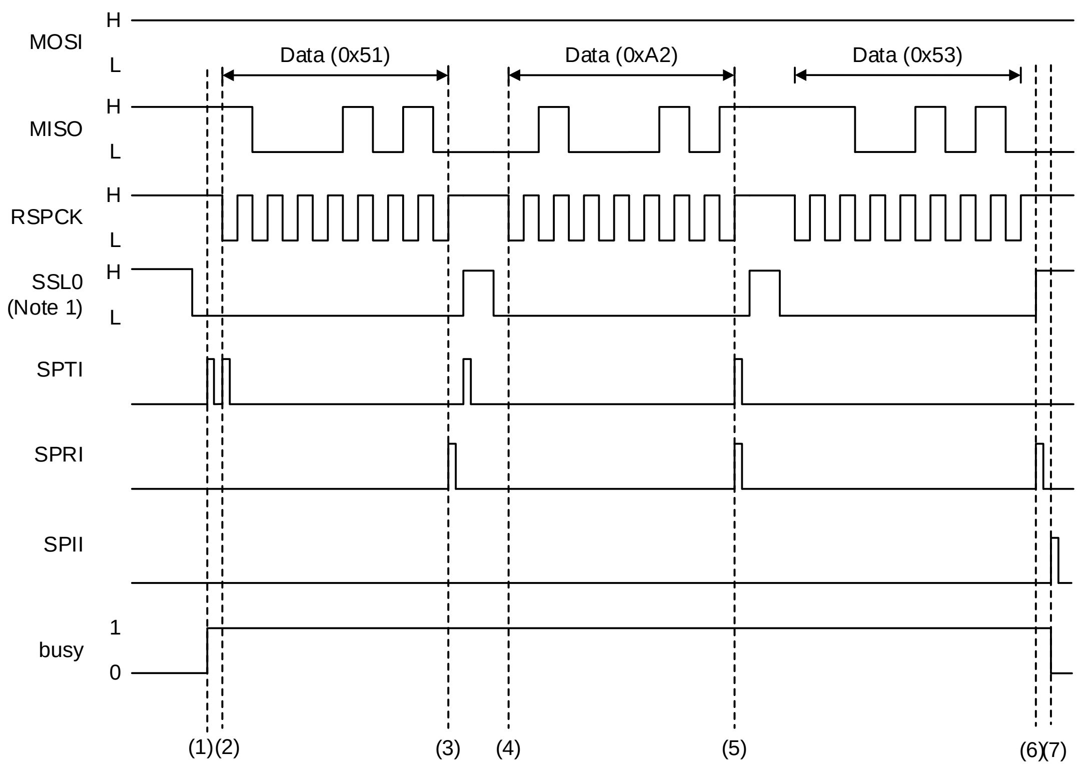
Figure 3-6 Reception Procedure in Master Mode

If a callback function has been set, it is called using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument when reception is completed.

When a reception error occurs, a callback function is called using ARM_SPI_EVENT_DATA_LOST as an argument and receive processing is completed.

The specific receive operation in master mode is different depending on what method is used for communication control: the interrupt, DMAC, or DTC. In the transmission-enabled state, dummy data is written to the transmit buffer in order that the clock be output. The dummy data to be output can be modified by the ARM_SPI_SET_DEFAULT_TX_VALUE command.

Figure 3-3 shows the operation when the interrupt is used for communication control. Figure 3-4 and Figure 3-5 respectively show the operations when the DMAC and the DTC are used for communication control.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-7 Receive Operation Using Interrupt for Control (3 bytes received, dummy data: 0xFF)

(1) When the Receive function is executed, the busy flag is set to “1” (communicating state). The SPTI interrupt is generated and dummy data is written to the data register (SPDR).

(2) At the second SPTI interrupt, the second byte of dummy data is written to the SPDR register.

(3) When data is received through the MISO pin, the SPRI interrupt occurs and the value of the data register (SPDR) is read into the specified buffer.

(4) At the SPTI interrupt generated when the specified number of bytes have been written, the SPTI interrupt is disabled, and the SPII interrupt is enabled.

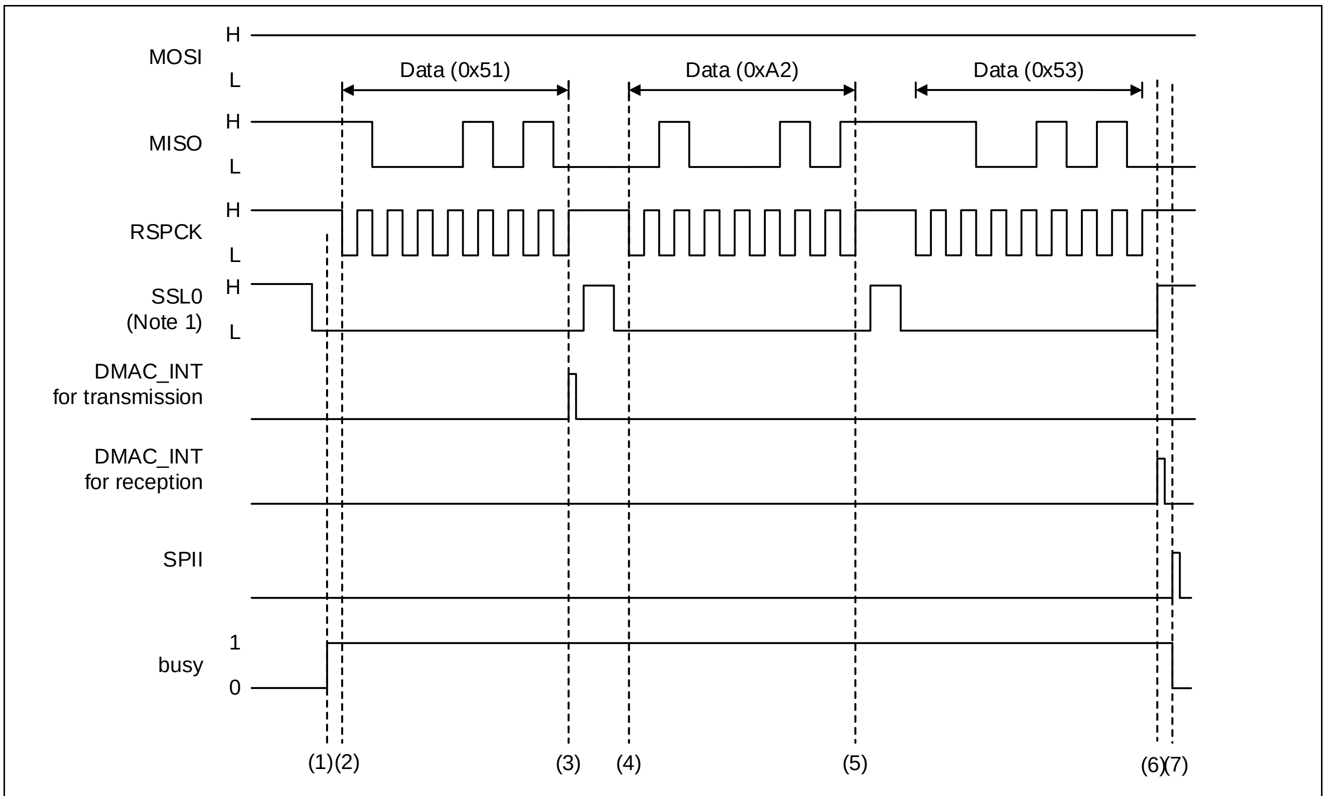
(5) The SPRI interrupt occurs upon completion of reception of each byte, and the received data is read from the SPDR register.

(6) Disable SPTI interrupt and enable SPII interrupt by SPTI interrupt by the output of the last dummy data written in (4).

(7) At the SPRI interrupt generated when the last data is read, the SPRI interrupt is disabled.

(8) When reception of all data is completed, the SPII interrupt occurs and the busy flag is cleared to “0” (communication wait state). In the SPII interrupt handling process, all the interrupts used for SPI control are disabled. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note)

Note: When a reception error occurs, the SPEI interrupt occurs, the busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-8 Receive Operation Using DMAC for Control (3 bytes received, dummy data: 0xFF)

(1) When the Receive function is executed, the busy flag is set to "1" (communicating state) and the SPTI and SPRI interrupts are set as DMAC transfer factors. Dummy data is DMA-transferred to the data register (SPDR).

(2) The second byte of the dummy data is DMA-transferred to the SPDR register.

(3) When data is received through the MISO pin, the value of the data register (SPDR) is DMA-transferred to the specified buffer.

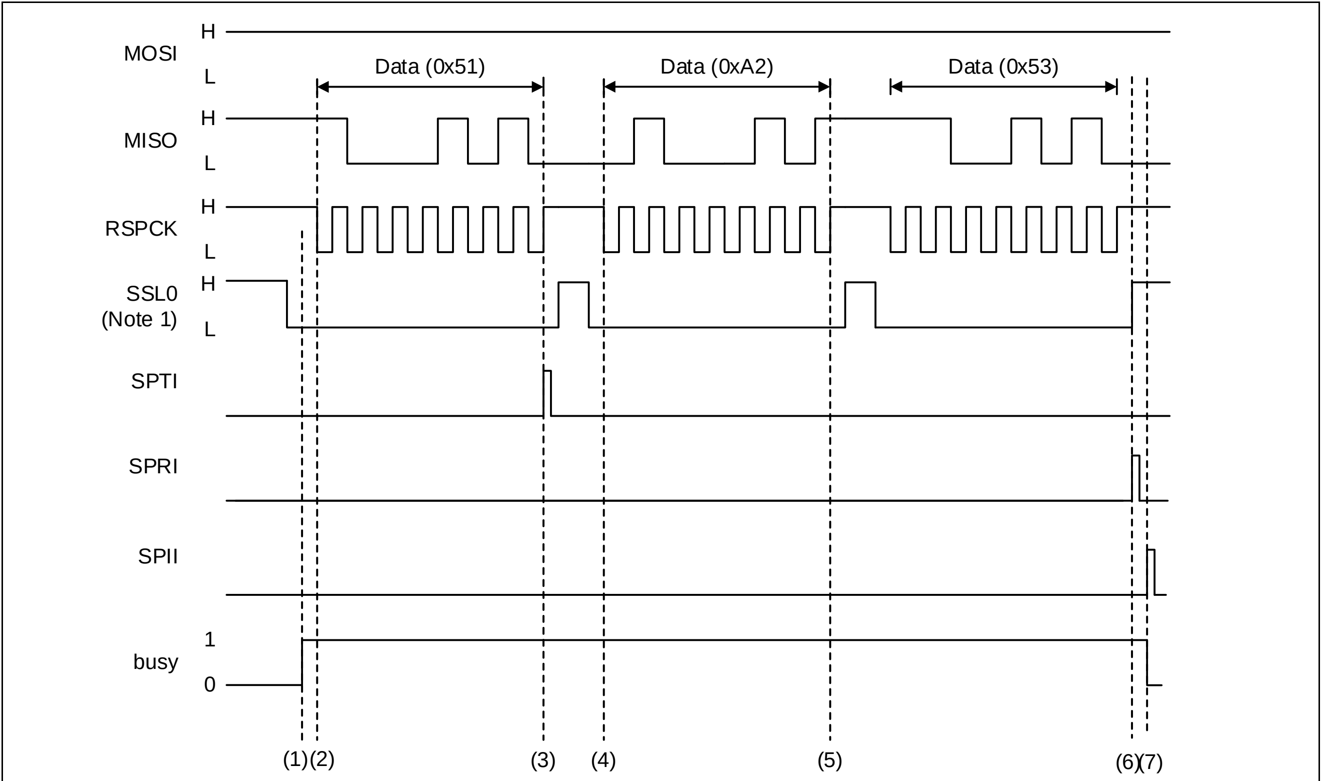
(4) At the DMAC transfer end interrupt generated when the specified number of bytes have been written, the DMAC transfer end interrupt is disabled, and the SPII interrupt is enabled.

(5) Each time reception of one byte is completed, the receive data in the SPDR register is DMA-transferred to the specified buffer.

(6) At the DMAC transfer end interrupt generated when reception of the final data has been completed, the DMAC transfer end interrupt is disabled.

(7) When reception of all data is completed, the SPII interrupt occurs and the busy flag is cleared to "0" (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note)

Note: When a reception error occurs, the SPEI interrupt occurs, the busy flag is cleared to "0" (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-9 Receive Operation Using DTC for Control (3 bytes received, dummy data: 0xFF)

(1) When the Receive function is executed, the busy flag is set to "1" (communicating state) and the SPTI and SPRI interrupts are set as DTC transfer factors. Dummy data is DMA-transferred to the data register (SPDR).

(2) The second byte of the dummy data is DMA-transferred to the SPDR register.

(3) When data is received through the MISO pin, the value of the data register (SPDR) is DMA-transferred to the specified buffer.

(4) At the SPTI interrupt generated when the specified number of bytes have been written, the SPTI interrupt is disabled, and the SPII interrupt is enabled.

(5) Each time reception of one byte is completed, the receive data in the SPDR register is DMA-transferred to the specified buffer.

(6) At the SPRI interrupt generated when reception of the last data has been completed, the SPRI interrupt is disabled.

(7) When reception of all data is completed, the SPII interrupt occurs and the busy flag is cleared to "0" (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note)

Note: When a reception error occurs, the SPEI interrupt occurs, the busy flag is cleared to "0" (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

3.1.4 Transmission and Reception in Master Mode

The transmission and reception procedure in master mode is shown in Figure 3-10.

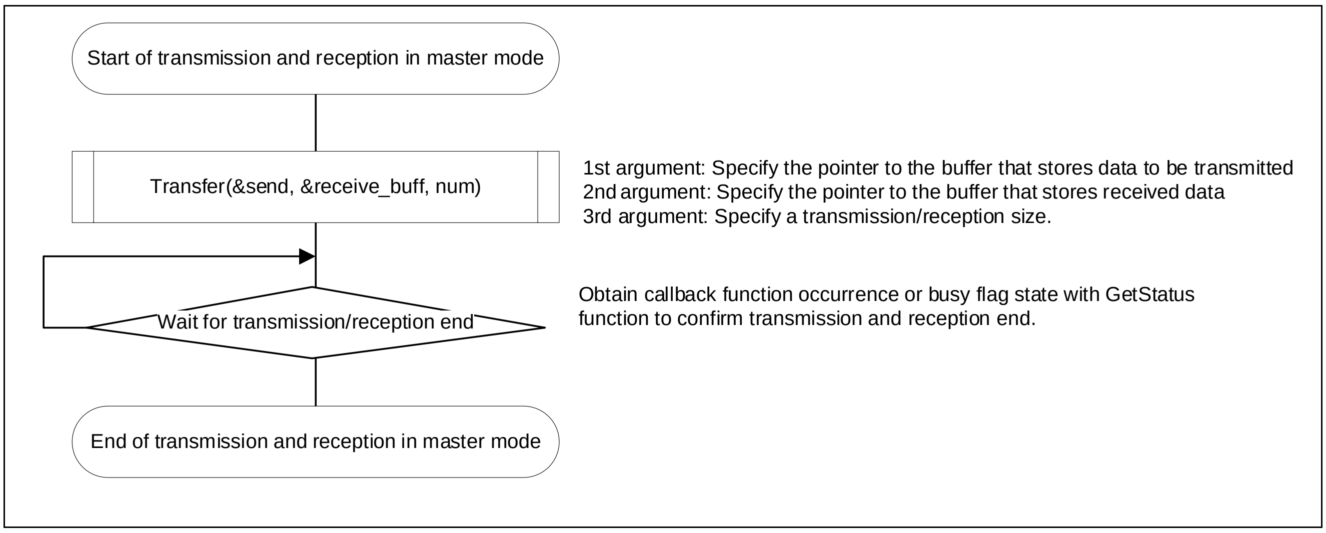
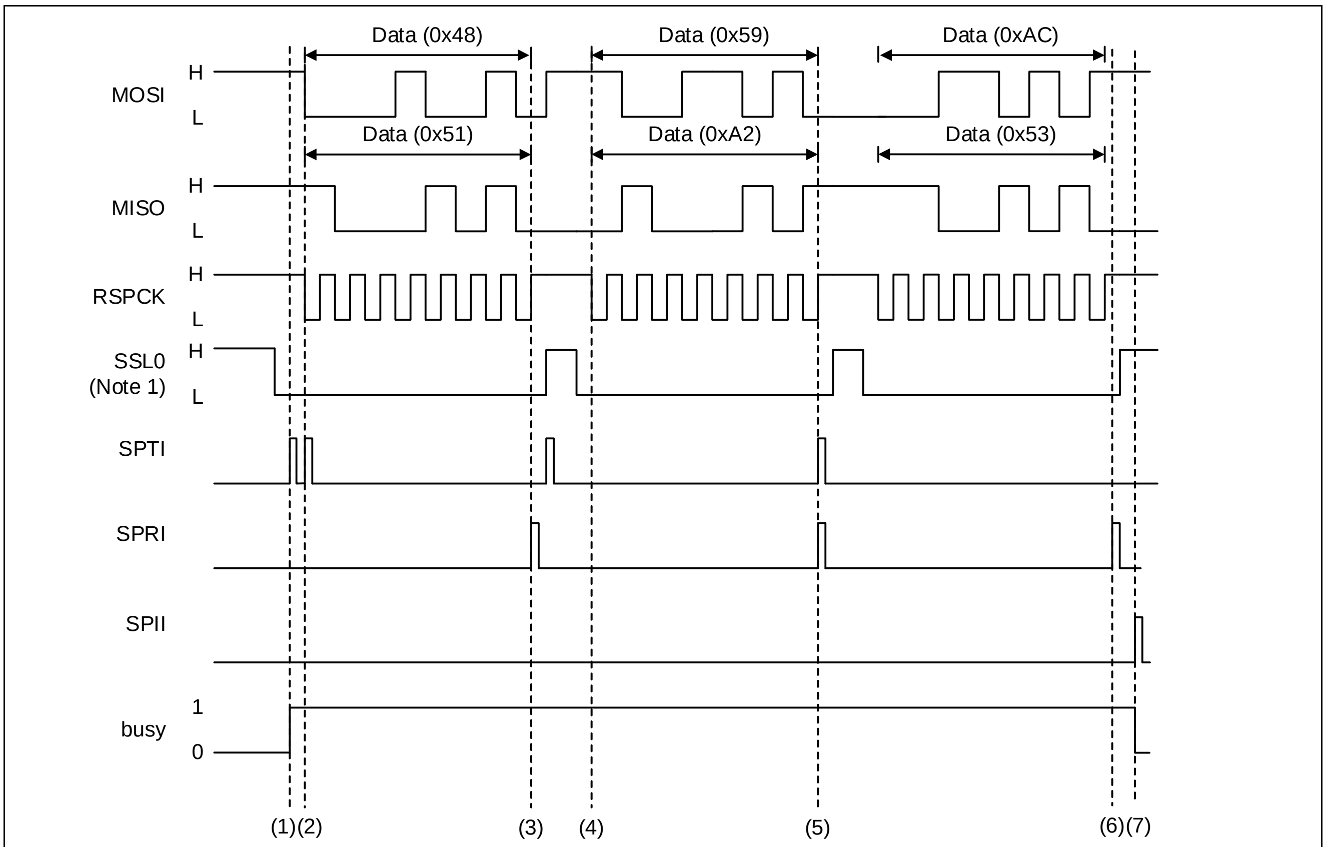

Start of transmission and reception in master mode
Transfer(&send, &receive_buff, num)
1st argument: Specify the pointer to the buffer that stores data to be transmitted
2nd argument: Specify the pointer to the buffer that stores received data
3rd argument: Specify a transmission/reception size.
Wait for transmission/reception end
Obtain callback function occurrence or busy flag state with GetStatus function to confirm transmission and reception end.
End of transmission and reception in master mode

Figure 3-10 Transmission and Reception Procedure in Master Mode

If a callback function has been set, it is called using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument when transmission/reception is completed.

When a transmission/reception error occurs, a callback function is called with ARM_SPI_EVENT_DATA_LOST as an argument and transmission and reception are terminated.

The specific transmit and receive operation in master mode is different depending on what method is used for communication control: the interrupt, DMAC, or DTC. Figure 3-3 shows the operation when the interrupt is used for communication control. Figure 3-4 and Figure 3-5 respectively show the operations when the DMAC and the DTC are used for communication control.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-11 Transmit and Receive Operation Using Interrupt for Control (3 bytes received)

(1) When the Transfer function is executed, the busy flag is set to “1” (communicating state). The SPTI interrupt is generated and transmit data is written to the data register (SPDR).

(2) At the second SPTI interrupt, the second byte of the transmit data is written to the SPDR register.

(3) When data is received through the MISO pin, the SPRI interrupt occurs and the value of the data register (SPDR) is read into the specified buffer.

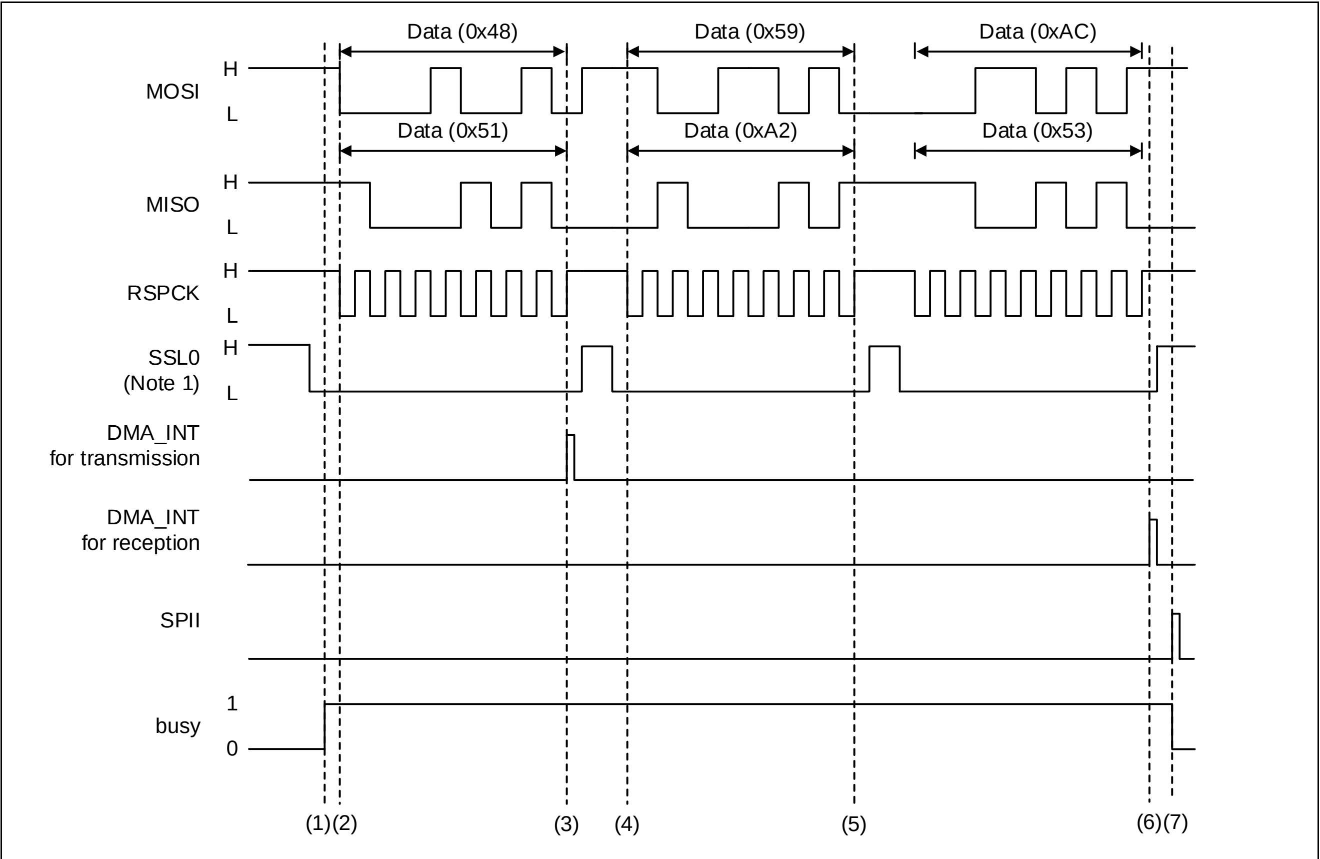
(4) At the SPTI interrupt generated when the specified number of bytes have been written, the SPTI interrupt is disabled, and the SPII interrupt is enabled.

(5) Each time reception of one byte is completed, the SPRI interrupt occurs and the received data is read from the SPDR register.

(6) At the SPRI interrupt generated when the last data is read, the SPRI interrupt is disabled.

(7) When reception is completed, the SPII interrupt occurs and the busy flag is cleared to “0” (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note)

Note: When a reception error occurs, the SPEI interrupt occurs, the busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-12 Transmit and Receive Operation Using DMAC for Control (3 bytes received)

(1) When the Transfer function is executed, the busy flag is set to “1” (communicating state) and the SPTI and SPRI interrupts are set as DMAC transfer factors. Dummy data is DMA-transferred to the data register (SPDR).

(2) The second byte of transmit data is DMA-transferred to the SPDR register.

(3) When data is received through the MISO pin, the value of the data register (SPDR) is DMA-transferred to the specified buffer.

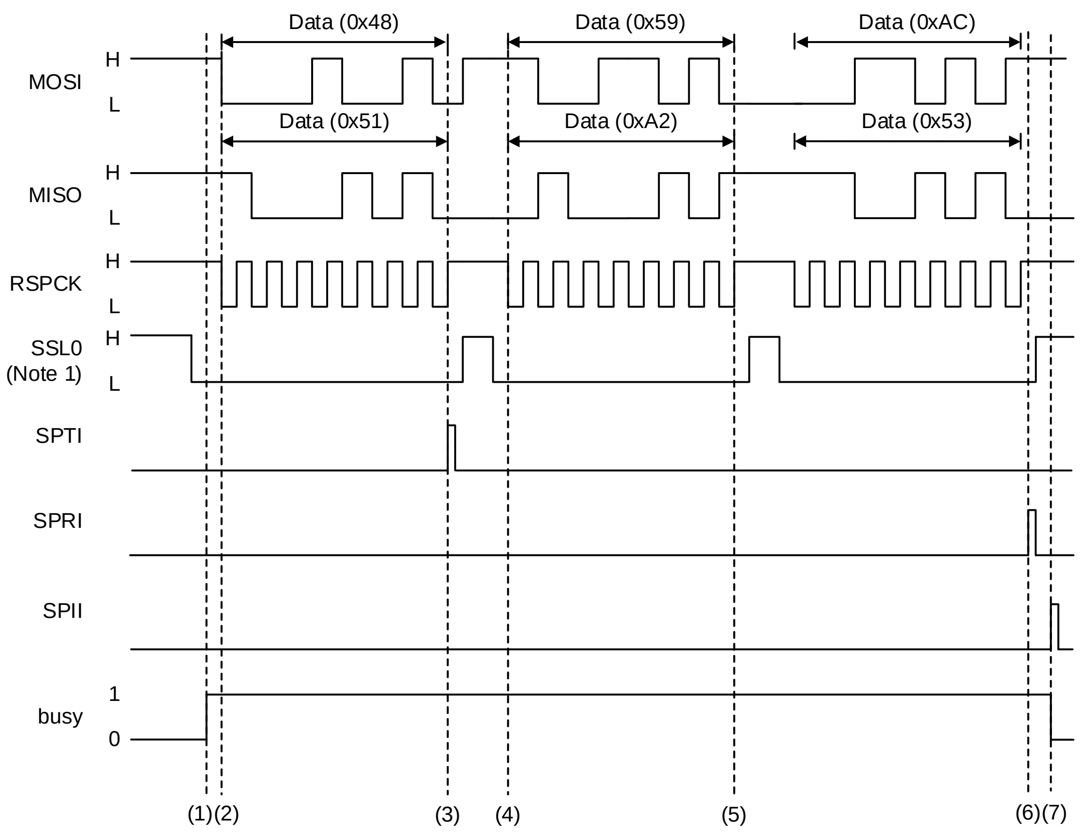
(4) At the DMAC transfer end interrupt generated when the specified number of bytes have been written, the DMAC transfer end interrupt is disabled, and the SPII interrupt is enabled.

(5) Each time reception of one byte is completed, the receive data in the SPDR register is DMA-transferred to the specified buffer.

(6) At the DMAC transfer end interrupt generated when reception of the last data has been completed, the DMAC transfer end interrupt is disabled.

(7) When reception is completed, the SPII interrupt occurs and the busy flag is cleared to “0” (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note)

Note: When a transmission/reception error occurs, the SPEI interrupt occurs, the busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-13 Transmit and Receive Operation Using DTC for Control (3 bytes received)

(1) When the Transfer function is executed, the busy flag is set to “1” (communicating state) and the SPTI and SPRI interrupts are set as DMAC transfer factors. Dummy data is DMA-transferred to the data register (SPDR).

(2) The second byte of transmit data is DMA-transferred to the SPDR register.

(3) When data is received through the MISO pin, the value of the data register (SPDR) is DMA-transferred to the specified buffer.

(4) At the SPTI interrupt generated when the specified number of bytes have been written, the SPTI interrupt is disabled, and the SPII interrupt is enabled.

(5) Each time reception of one byte is completed, the receive data in the SPDR register is DMA-transferred to the specified buffer.

(6) At the SPRI interrupt generated when reception of the last data has been completed, the SPRI interrupt is disabled.

(7) When reception is completed, the SPII interrupt occurs and the busy flag is cleared to “0” (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note)

Note: When a transmission/reception error occurs, the SPEI interrupt occurs, the busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

3.2 Slave Mode

3.2.1 Initial Setting Procedure for Slave Mode

The initial setting procedure for slave mode is shown in Figure 3-14.

When enabling transmission and reception, register the interrupts to use with NVIC in r_system_cfg.h. For details, see section 2.4, Communication Control and NVIC Interrupt Setting.

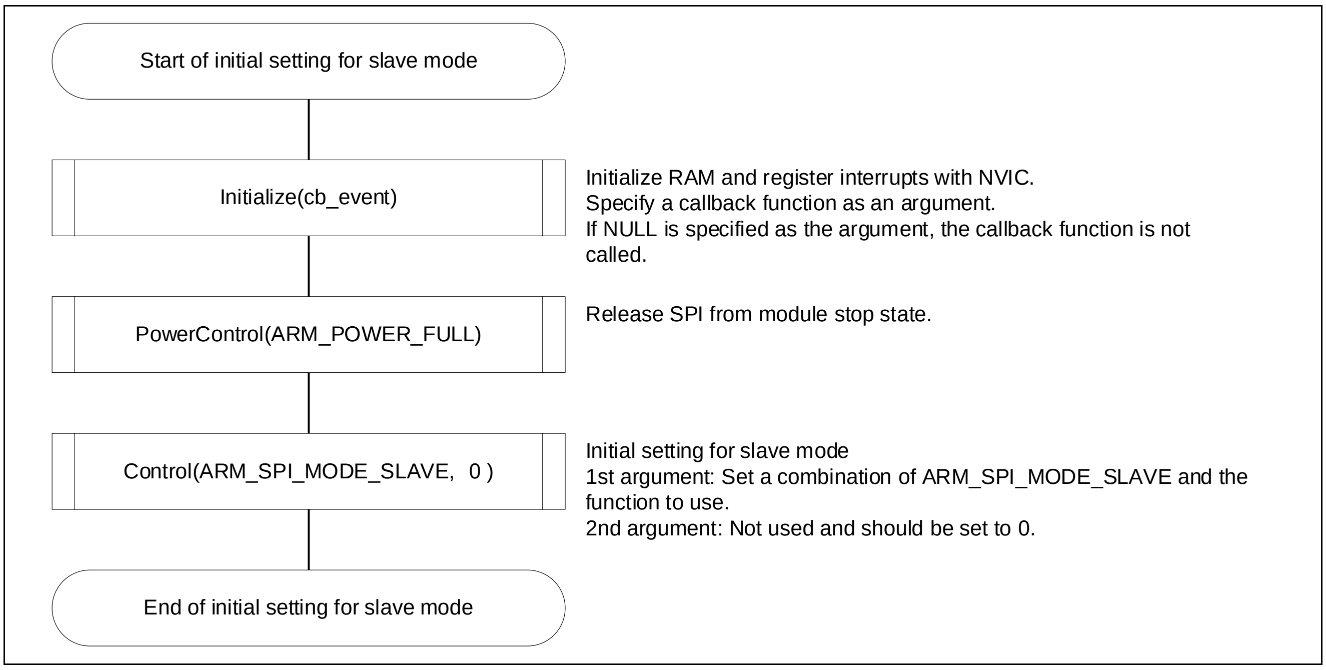

Start of initial setting for slave mode
Initialize(cb_event)
Initialize RAM and register interrupts with NVIC.
Specify a callback function as an argument.
If NULL is specified as the argument, the callback function is not called.
PowerControl(ARM_POWER_FULL)
Release SPI from module stop state.
Control(ARM_SPI_MODE_SLAVE, 0)
Initial setting for slave mode
1st argument: Set a combination of ARM_SPI_MODE_SLAVE and the function to use.
2nd argument: Not used and should be set to 0.
End of initial setting for slave mode

Figure 3-14 Slave Mode Initialization Procedure

3.2.2 Transmission in Slave Mode

The transmission procedure in slave mode is shown in Figure 3-15.

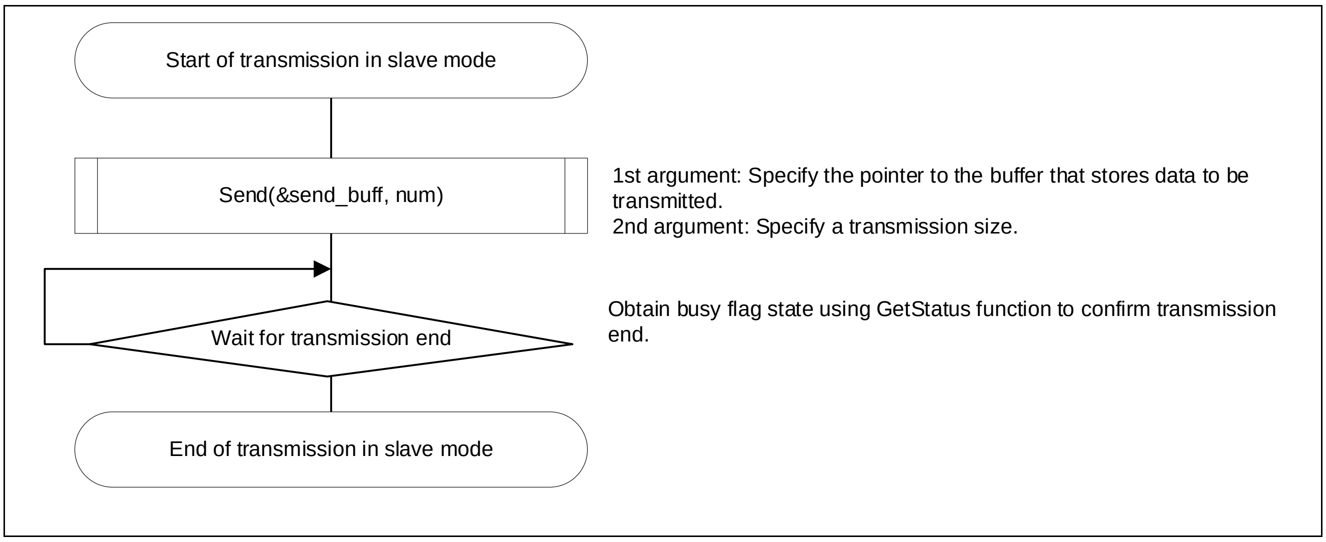
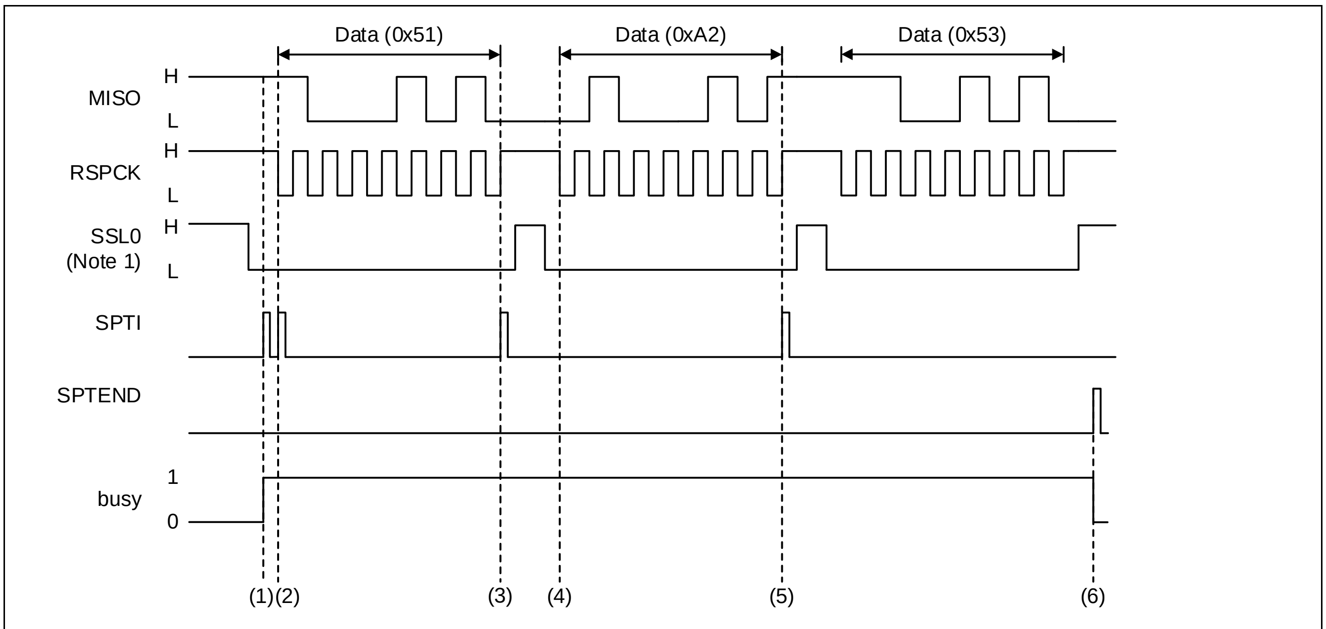

Start of transmission in slave mode
Send(&send_buff, num)
1st argument: Specify the pointer to the buffer that stores data to be transmitted.
2nd argument: Specify a transmission size.
Wait for transmission end
Obtain busy flag state using GetStatus function to confirm transmission end.
End of transmission in slave mode

Figure 3-15 Transmission Procedure in Slave Mode

If a callback function has been set, it is called using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument when transmission is completed.

When a transmission error occurs, a callback function is called with ARM_SPI_EVENT_DATA_LOST as an argument and the transmission is terminated.

The specific transmit operation in slave mode is different depending on what method is used for communication control: the interrupt, DMAC, or DTC. Figure 3-3 shows the transmit operation when the interrupt is used for communication control. Figure 3-4 and Figure 3-5 respectively show the operations when the DMAC and the DTC are used for communication control.



busy: Communication state (can be obtained with GetStatus function)

Note 1: SSL pin input is enabled only when SSL pin input monitoring with hardware is specified (ARM_SPI_SS_SLAVE_HW).

Figure 3-16 Transmit Operation Using Interrupt for Control (3 bytes transmitted)

(1) When the Send function is executed, the busy flag is set to “1” (communicating state). The SPTI interrupt is generated and the first byte of data is written to the transmit data register (SPDR).

(2) When the clock is input to the RSPCK pin, the first byte of data starts to be output from the MOSI pin, and the second SPTI interrupt occurs. In the interrupt handling processing, the second byte of transmit data is written to the SPDR register.

(3) At the third SPTI interrupt, the last transmit data is written to SPDR.

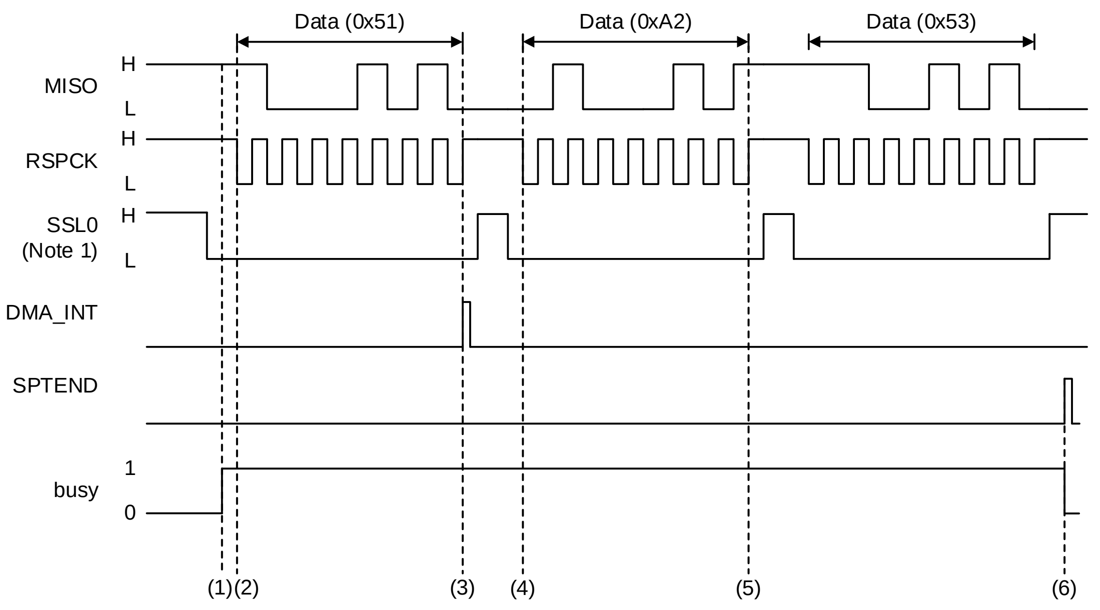
(4) At the SPTI interrupt generated after the last data has been written, the SPTI interrupt is disabled, and the SPTEND interrupt is enabled.

(5) After the second byte of the data is output, the last data written in step 3 is output.

(6) When transmission is completed, the SPTEND interrupt occurs and the busy flag is cleared to “0” (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note1) (Note2)

Note 1. When a transmission error occurs, the SPEI interrupt occurs, the busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

Note 2. When CPHA = 0 is set, wait for half a RSPCK cycle in software before resuming communication. For details, see “5.6 Resuming communication in slave mode and CPHA0”.



busy: Communication state (can be obtained with GetStatus function)

Note 1: SSL pin input is enabled only when SSL pin input monitoring with hardware is specified (ARM_SPI_SS_SLAVE_HW).

Figure 3-17 Transmit Operation Using DMAC for Control (3 bytes transmitted)

(1) When the Send function is executed, the busy flag is set to “1” (communicating state) and the SPTI interrupt is set as a DMAC transfer factor. The first byte of transmit data is DMA-transferred to the SPDR register.

(2) When the clock is input to the RSPCK pin, the first byte of data starts to be output from the MOSI pin, and the second byte of the transmit data is transferred to the SPDR register by the second DMA transfer.

(3) The last transmit data is written to the SPDR register by the third DMA transfer.

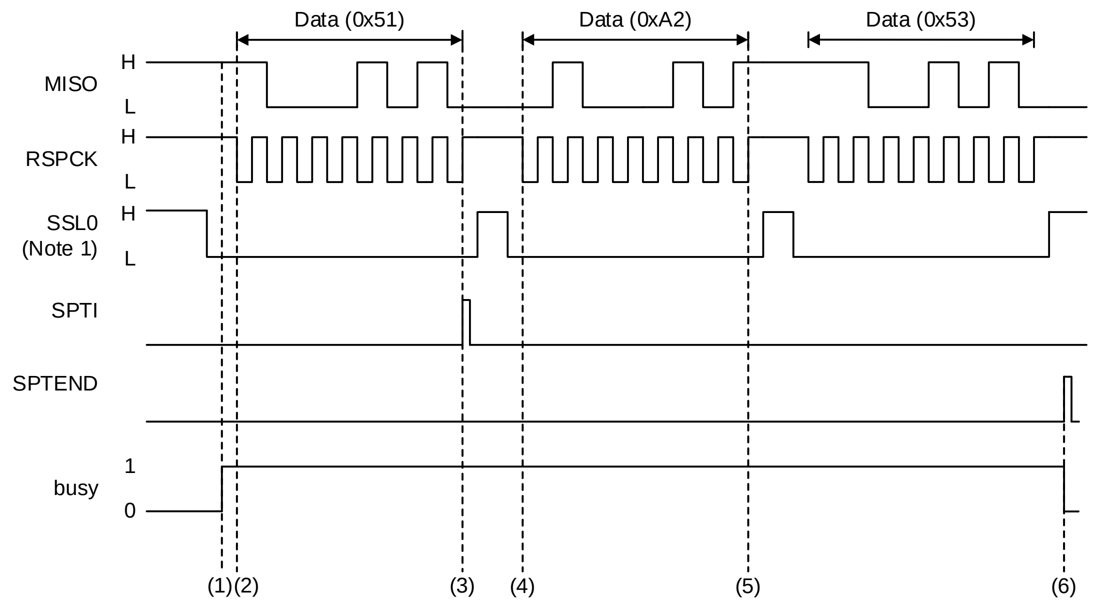
(4) At the DMAC transfer end interrupt generated after the last data has been written, the DMAC transfer end interrupt is disabled, and the SPTEND interrupt is enabled.

(5) After the second byte of the data is output, the last data written in step 3 is output.

(6) When transmission is completed, the SPTEND interrupt occurs and the busy flag is cleared to “0” (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note1) (Note2)

Note 1. If a transmission error occurs, the SPEI interrupt occurs, the busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

Note 2. When CPHA = 0 is set, wait for half a RSPCK cycle in software before resuming communication. For details, see “5.6 Resuming communication in slave mode and CPHA0”.



busy: Communication state (can be obtained with GetStatus function)

Note 1: SSL pin input is enabled only when SSL pin input monitoring with hardware is specified (ARM_SPI_SS_SLAVE_HW).

Figure 3-18 Transmit Operation Using DTC for Control (3 bytes transmitted)

(1) When the Send function is executed, the busy flag is set to "1" (communicating state) and the SPTI interrupt is set as a DTC transfer factor. The first byte of transmit data is DMA-transferred to the SPDR register.

(2) When the clock is input to the RSPCK pin, the first byte of data starts to be output from the MISO pin, and the second byte of the transmit data is transferred to the SPDR register by the second DMA transfer.

(3) The last transmit data is written to the SPDR register by the third DMA transfer.

(4) At the SPTI interrupt generated after the last data has been written, the SPTI interrupt is disabled, and the SPTEND interrupt is enabled.

(5) After the second byte of the data is output, the last data written in step 3 is output.

(6) When transmission is completed, the SPTEND interrupt occurs and the busy flag is cleared to "0" (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note)

Note: If a transmission error occurs, the SPEI interrupt occurs, the busy flag is cleared to "0" (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

3.2.3 Reception in Slave Mode

The reception procedure in slave mode is shown in Figure 3-19.

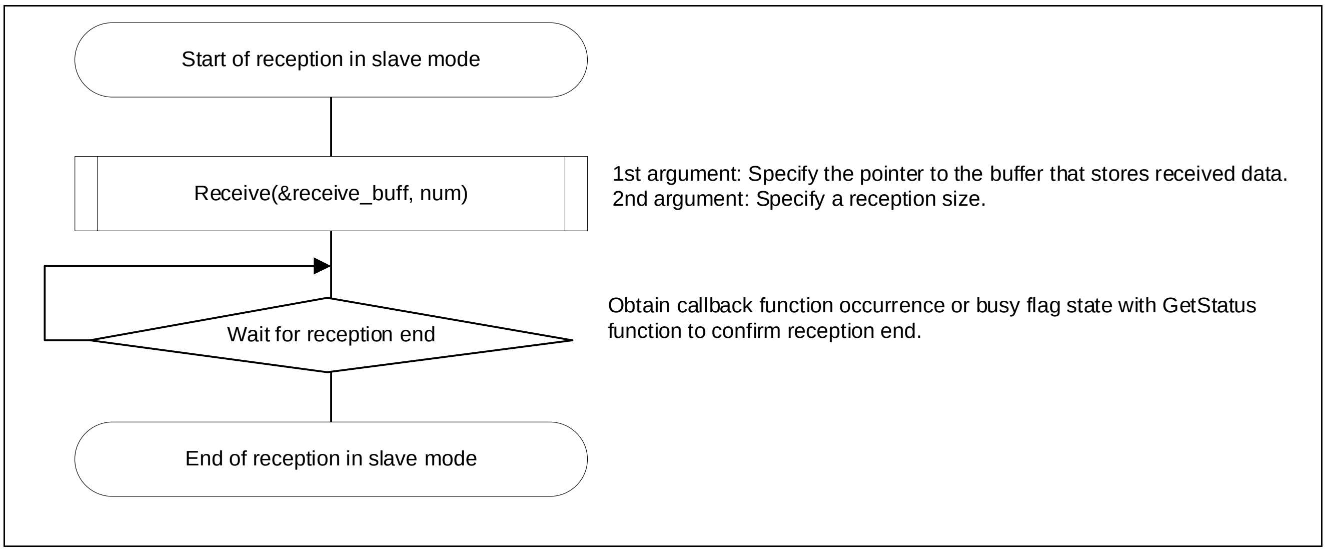

Start of reception in slave mode
Receive(&receive_buff, num)
1st argument: Specify the pointer to the buffer that stores received data.
2nd argument: Specify a reception size.
Wait for reception end
Obtain callback function occurrence or busy flag state with GetStatus function to confirm reception end.
End of reception in slave mode

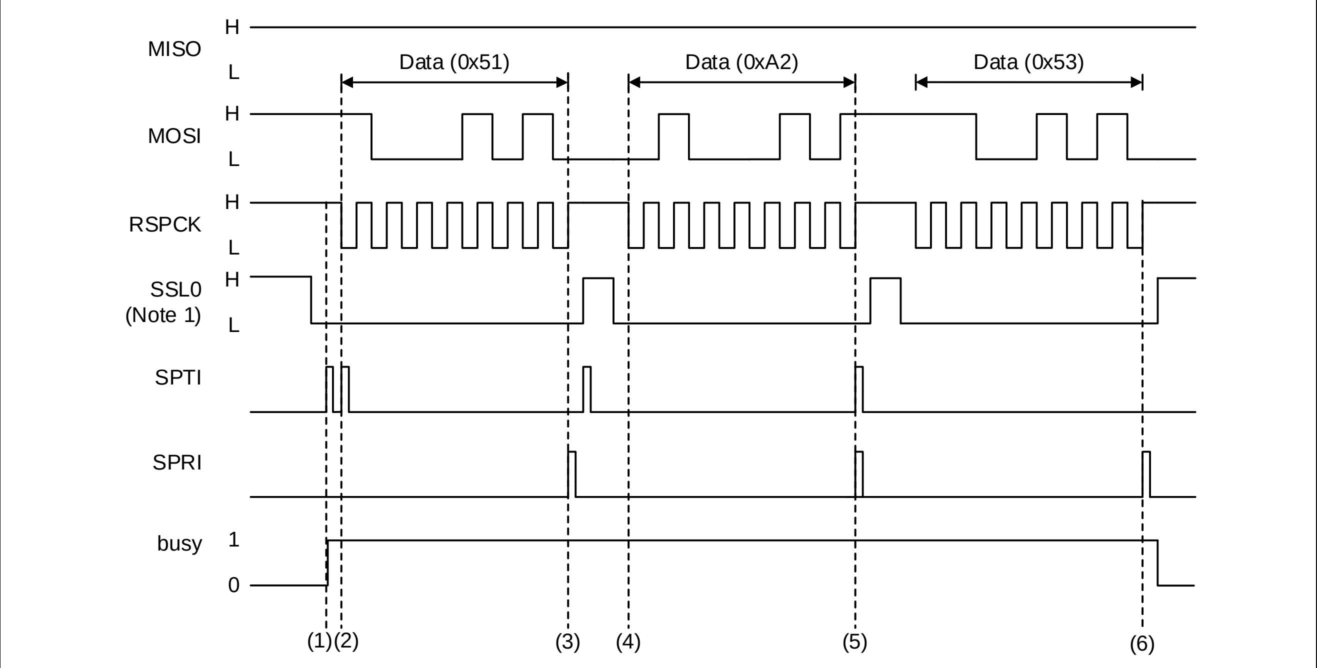
Figure 3-19 Reception Procedure in Slave Mode

If a callback function has been set, it is called using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument when reception is completed.

When a reception error occurs, a callback function is called with ARM_SPI_EVENT_DATA_LOST as an argument and receive processing is terminated.

The specific receive operation in slave mode is different depending on what method is used for communication control: the interrupt, DMAC, or DTC. In the transmission-enabled state, dummy data is written to the transmit buffer. The dummy data to be output can be modified by the ARM_SPI_SET_DEFAULT_TX_VALUE command.

Figure 3-3 shows the operation when the interrupt is used for communication control. Figure 3-4 and Figure 3-5 respectively show the operations when the DMAC and the DTC are used for communication control.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-20 Receive Operation Using Interrupt for Control (3 bytes received, dummy data: 0xFF)

(1) When the Receive function is executed, the busy flag is set to "1" (communicating state). The SPTI interrupt is generated and dummy data is written to the data register (SPDR).

(2) When the clock is input to the RSPCK pin, the first byte of dummy data starts to be output from the MISO pin, and the second byte of the dummy data is written to the SPDR register at the second SPTI interrupt.

(3) When data is received through the MOSI pin, the SPRI interrupt occurs and the value of the data register (SPDR) is read into the specified buffer.

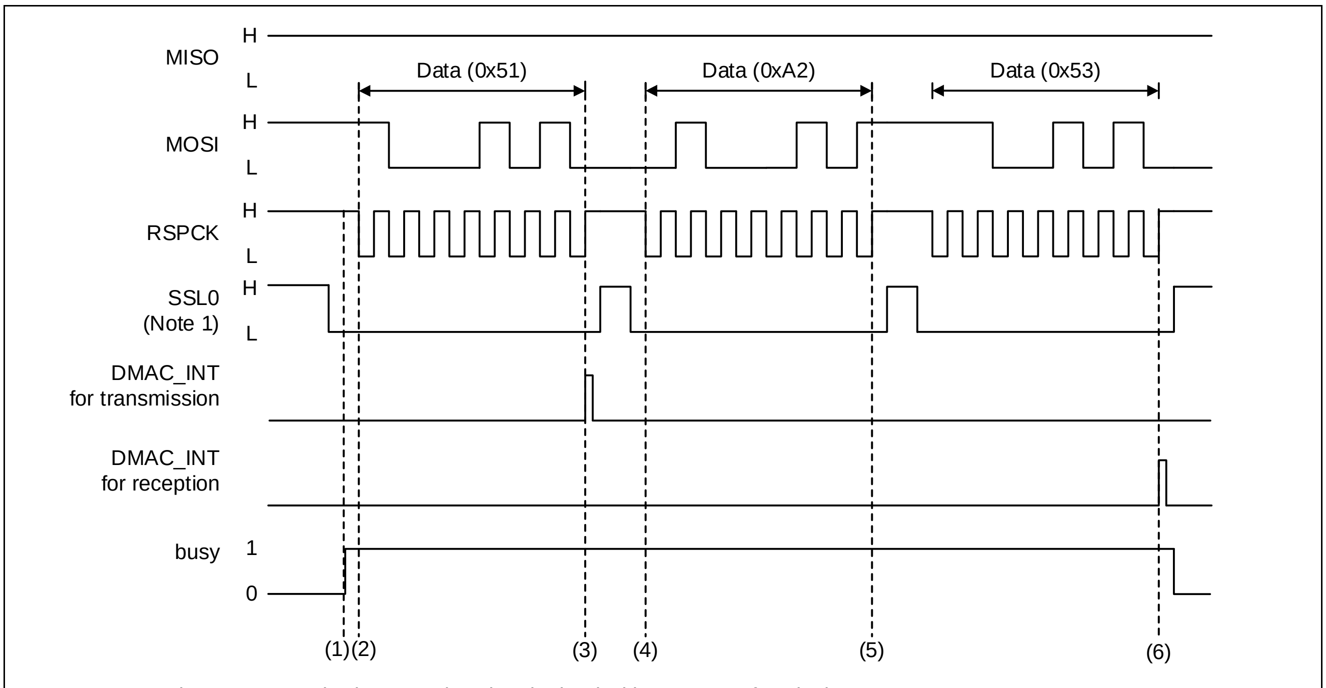
(4) At the SPTI interrupt generated when the specified number of bytes have been written, the SPTI interrupt is disabled, and the SPII interrupt is enabled.

(5) The SPRI interrupt occurs upon completion of reception of each byte, and the received data is read from the SPDR register.

(6) At the SPRI interrupt generated when the last data is read, the SPRI interrupt is disabled. The busy flag is cleared to "0" (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note1) (Note2)

Note 1. When a transmission/reception error occurs, the SPEI interrupt occurs, the busy flag is cleared to "0" (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

Note 2. When CPHA = 0 is set, wait for half a RSPCK cycle in software before resuming communication. For details, see "5.6 Resuming communication in slave mode and CPHA0".



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-21 Receive Operation Using DMAC for Control (3 bytes received, dummy data: 0xFF)

(1) When the Receive function is executed, the busy flag is set to “1” (communicating state) and the SPTI and SPRI interrupts are set as DMAC transfer factors. Dummy data is DMA-transferred to the data register (SPDR).

(2) When the clock is input to the RSPCK pin, dummy data starts to be output from the MOSI pin, and the second and subsequent bytes of dummy data are DMA-transferred to the transmit data register (SPDR).

(3) When data is received through the MOSI pin, the value of the receive data register (SPDR) is DMA-transferred to the specified buffer.

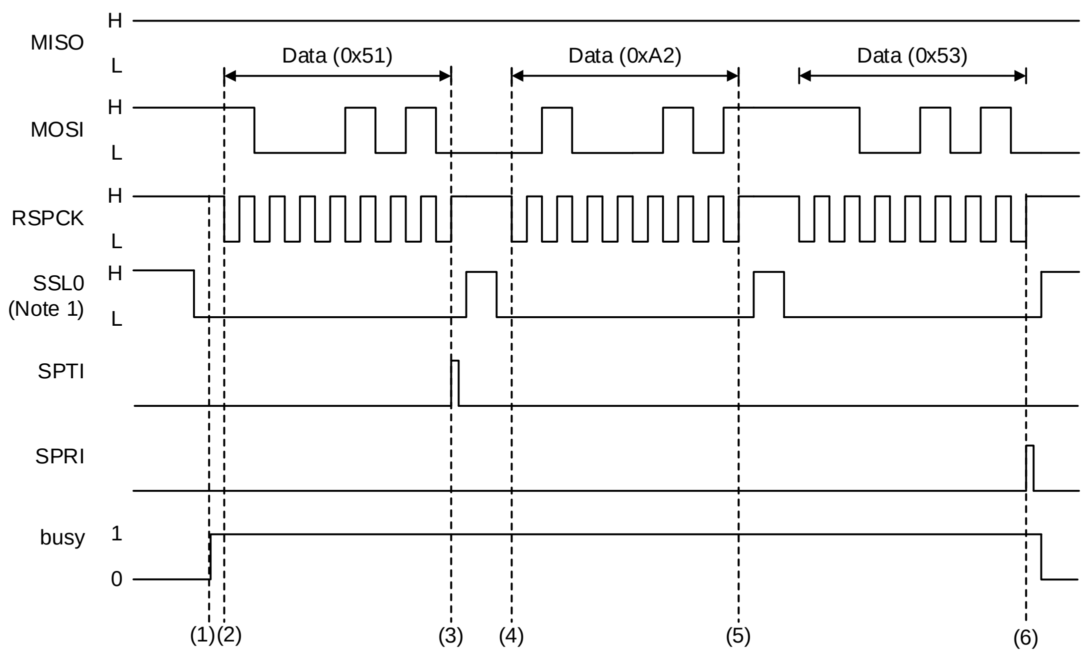
(4) When the specified number of bytes have been written, a DMAC transfer end interrupt occurs on the transmission side.

(5) Each time reception of one byte is completed, the value of the SPDR register is DMA-transferred to the specified buffer.

(6) After transfer of the specified size of data is completed, the DMAC transfer end interrupt occurs on the transmitting side. The rx_busy flag is cleared to "0" (communication wait state) in the interrupt handling process. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note1) (Note2)

Note 1. When a reception error occurs, the SPEI interrupt occurs, the rx_busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

Note 2. When CPHA = 0 is set, wait for half a RSPCK cycle in software before resuming communication. For details, see “5.6 Resuming communication in slave mode and CPHA0”.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-22 Receive Operation Using DTC for Control (3 bytes received, dummy data: 0xFF)

(1) When the Receive function is executed, the busy flag is set to “1” (communicating state) and the SPTI and SPRI interrupts are set as DTC transfer factors. Dummy data is DMA-transferred to the data register (SPDR).

(2) When the clock is input to the RSPCK pin, dummy data starts to be output from the MOSI pin, and the second and subsequent bytes of dummy data are DMA-transferred to the transmit data register (SPDR).

(3) When data is received through the MOSI pin, the value of the receive data register (SPDR) is DMA-transferred to the specified buffer.

(4) When the specified number of bytes have been written, the DMAC transfer end interrupt occurs on the transmission side.

(5) Each time reception of one byte is completed, the value of the SPDR register is DMA-transferred to the specified buffer.

(6) After transfer of the specified size of data is completed, the DMAC transfer end interrupt occurs. The rx_busy flag is cleared to "0" (communication wait state) in the interrupt handling process. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note1) (Note2)

Note 1. If a reception error occurs, the SPEI interrupt occurs, the rx_busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

Note 2. When CPHA = 0 is set, wait for half a RSPCK cycle in software before resuming communication. For details, see “5.6 Resuming communication in slave mode and CPHA0”.

3.2.4 Transmission and Reception in Slave Mode

The transmission and reception procedure in slave mode is shown inFigure 3-23.

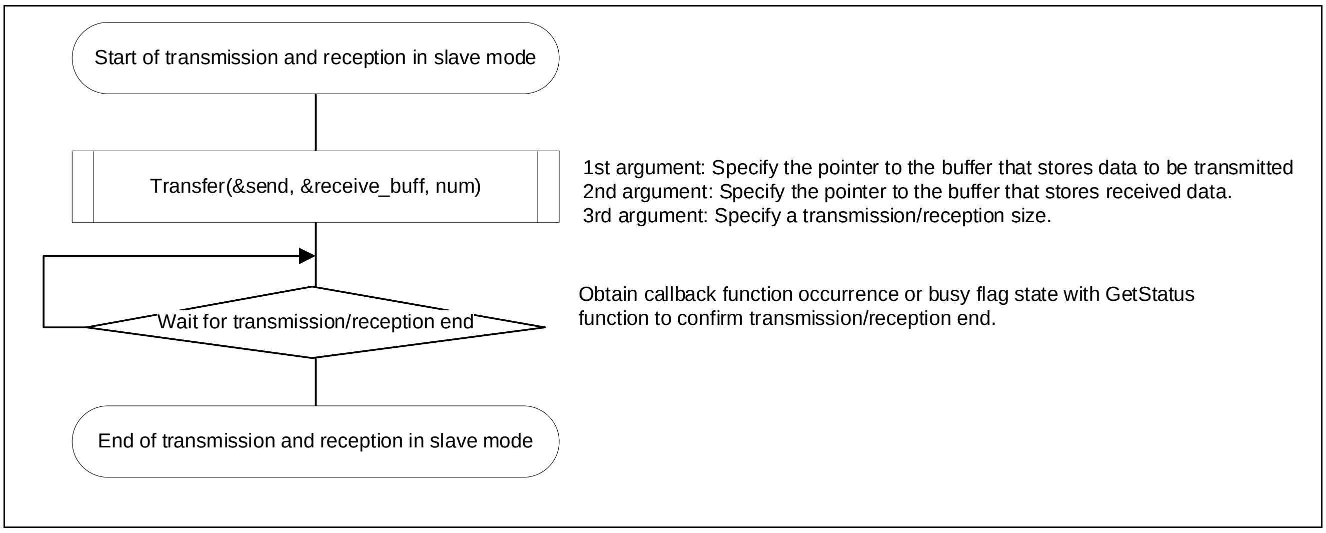

Start of transmission and reception in slave mode
Transfer(&send, &receive_buff, num)
1st argument: Specify the pointer to the buffer that stores data to be transmitted
2nd argument: Specify the pointer to the buffer that stores received data.
3rd argument: Specify a transmission/reception size.
Wait for transmission/reception end
Obtain callback function occurrence or busy flag state with GetStatus function to confirm transmission/reception end.
End of transmission and reception in slave mode

Figure 3-23 Transmission and Reception Procedure in Slave Mode

If a callback function has been set, the callback function is called using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument when reception is completed.

If a transmission/reception error occurs, a callback function is called with ARM_SPI_EVENT_DATA_LOST as an argument and transmit/receive processing is terminated.

The specific transmit and receive operation in slave mode is different depending on what method is used for communication control: the interrupt, DMAC, or DTC. Figure 3-3 shows the operation when the interrupt is used for communication control. Figure 3-4 and Figure 3-5 respectively show the operations when the DMAC and the DTC are used for communication control.

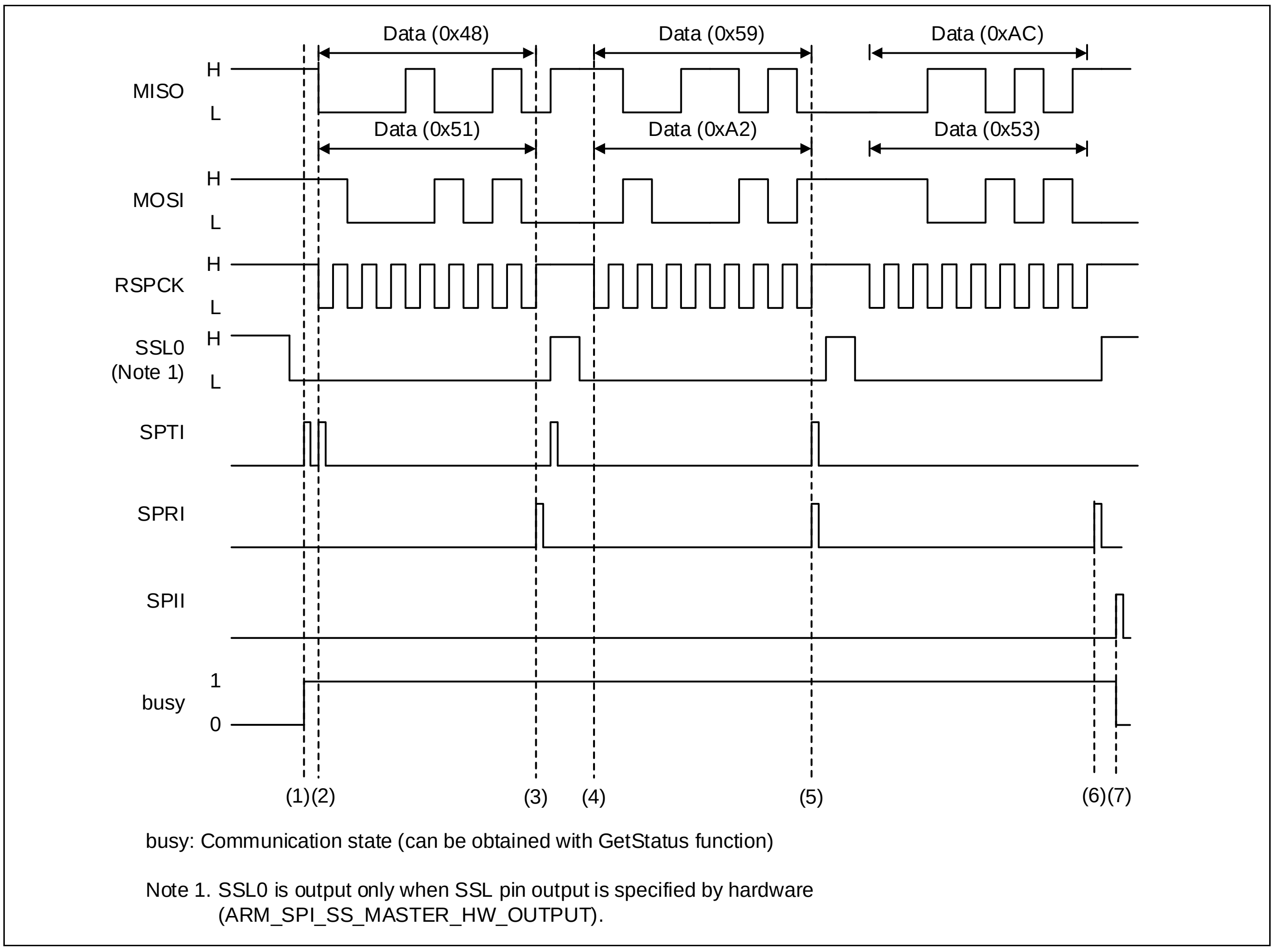

Data (0x48)
Data (0x59)
Data (0xAC)
MISO
H
L
Data (0x51)
Data (0xA2)
Data (0x53)
MOSI
RSPCK
SSL0 (Note 1)
SPTI
SPRI
SPII
busy
1
0
(1)(2)
(3)
(4)
(5)
(6)(7)
busy: Communication state (can be obtained with GetStatus function)
Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-24 Transmit and Receive Operation Using Interrupt for Control (3 bytes received)

(1) When the Transfer function is executed, the busy flag is set to “1” (communicating state). The SPTI interrupt is generated and the first transmit data is written to the data register (SPDR).

(2) When the clock is input to the RSPCK pin, the first byte of dummy data starts to be output from the MISO pin, and the second byte of the transmit data is written to the SPDR register at the second SPTI interrupt.

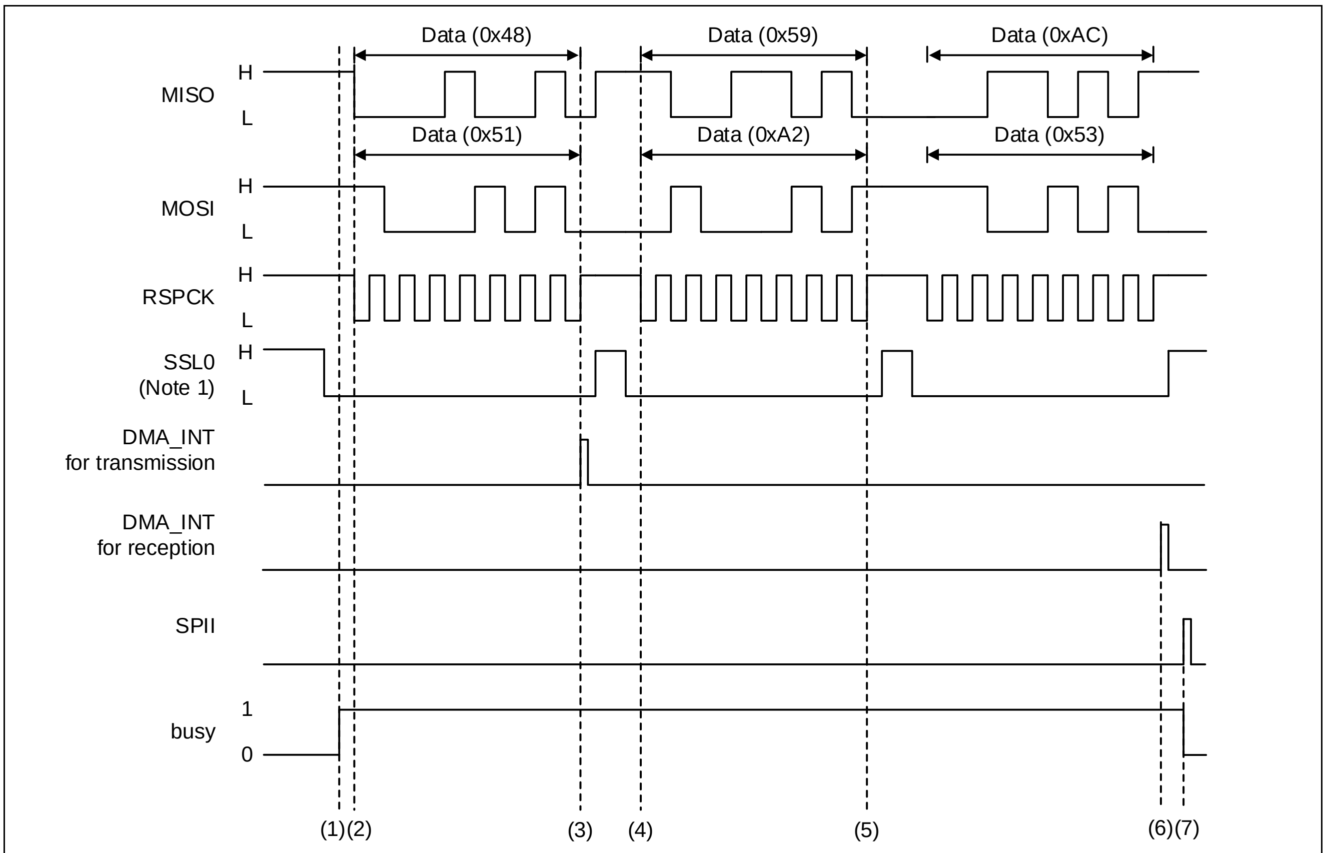
(3) When data is received through the MOSI pin, the SPRI interrupt occurs and the value of the data register (SPDR) is read into the specified buffer.

(4) At the SPTI interrupt generated when the specified number of bytes have been written, the SPTI interrupt is disabled, and the SPII interrupt is enabled.

(5) Each time reception of one byte is completed, the SPRI interrupt occurs and the received data is read from the SPDR register.

(6) At the SPRI interrupt generated when the last data has been read, the SPRI interrupt is disabled. The busy flag is cleared to “0” (communication wait state). Also disables all interrupts used for SPI control. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note)

Note: If a transmission/reception error occurs, the SPEI interrupt occurs, the busy flag is cleared to “0” (communication wait state), and the error state is cleared. Also disables all interrupts used for SPI control. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-25 Transmit and Receive Operation Using DMAC for Control (3 bytes received)

(1) When the Transfer function is executed, the busy flag is set to “1” (communicating state) and the SPTI and SPRI interrupts are set as DMAC transfer factors. Transmit data is DMA-transferred to the data register (SPDR).

(2) When the clock is input to the RSPCK pin, the transmit data starts to be output from the MOSI pin, and the second and subsequent bytes of the transmit data are DMA-transferred to the transmit data register (SPDR).

(3) When data is received through the MOSI pin, the value of the receive data register (SPDR) is DMA-transferred to the specified buffer.

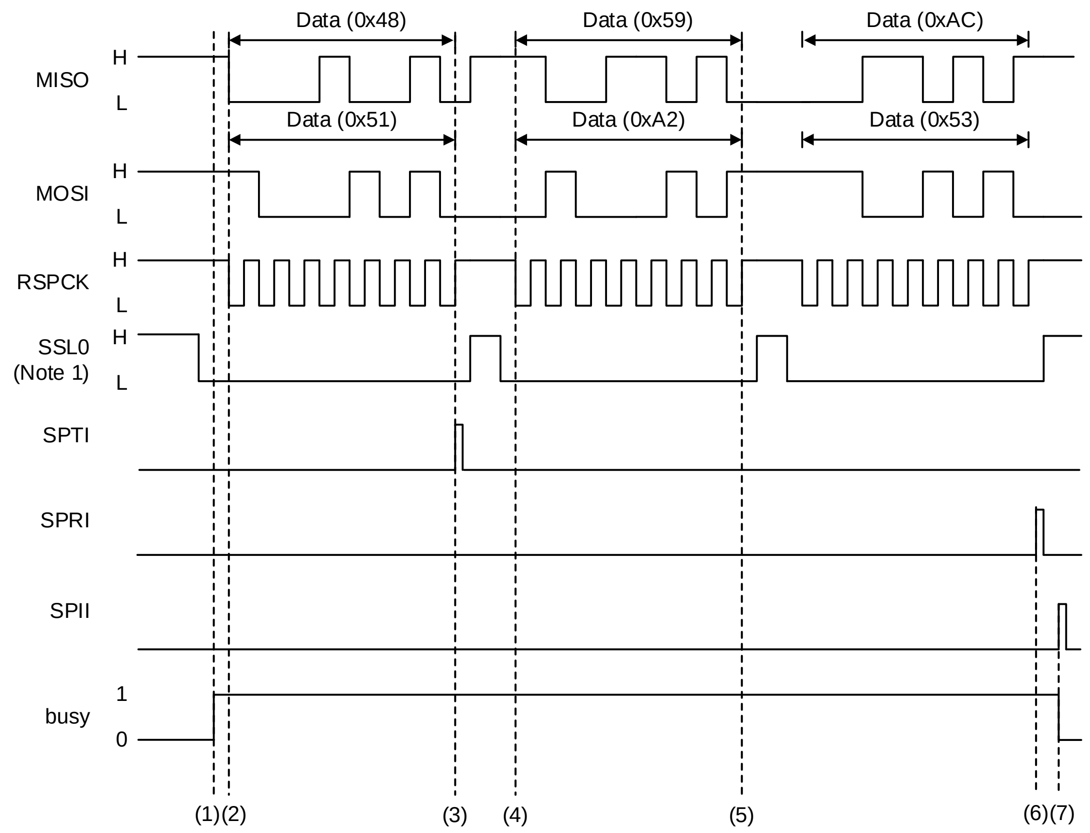
(4) When the specified number of bytes have been written, a DMAC transfer end interrupt occurs on the transmission side.

(5) Each time reception of one byte is completed, the value of the SPDR register is transferred to the specified buffer.

(6) After transfer of the specified size of data is completed, the DMAC transfer end interrupt occurs. The rx_busy flag is cleared to "0" (communication wait state) in the interrupt handling process. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note1) (Note2)

Note: If a transmission/reception error occurs, the SPEI interrupt occurs, the rx_busy flag is cleared to “0” (communication wait state), and the error state is cleared. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

Note 2. When CPHA = 0 is set, wait for half a RSPCK cycle in software before resuming communication. For details, see “5.6 Resuming communication in slave mode and CPHA0”.



busy: Communication state (can be obtained with GetStatus function)

Note 1. SSL0 is output only when SSL pin output is specified by hardware (ARM_SPI_SS_MASTER_HW_OUTPUT).

Figure 3-26 Transmit and Receive Operation Using DTC for Control (3 bytes received)

(1) When the Transfer function is executed, the busy flag is set to “1” (communicating state) and the SPTI and SPRI interrupts are set as DTC transfer factors. Transmit data is DMA-transferred to the data register (SPDR).

(2) When the clock is input to the RSPCK pin, the transmit data starts to be output from the MOSI pin, and the second and subsequent bytes of the transmit data are DMA-transferred to the transmit data register (SPDR).

(3) When data is received through the MOSI pin, the value of the receive data register (SPDR) is DMA-transferred to the specified buffer.

(4) When the specified number of bytes have been written, a DMAC transfer end interrupt occurs on the transmission side.

(5) Each time reception of one byte is completed, the value of the SPDR register is transferred to the specified buffer.

(6) After transfer of the specified data size is completed, the DMAC transfer end interrupt occurs. The rx_busy flag is cleared to "0" (communication wait state) in the interrupt handling process. If a callback function has been registered, the callback function is executed using ARM_SPI_EVENT_TRANSFER_COMPLETE as an argument. (Note1) (Note2)

Note: If a reception error occurs, the SPEI interrupt occurs, the rx_busy flag is cleared to “0” (communication wait state), and the error state is cleared. When a callback function is registered, the callback function is executed with ARM_SPI_EVENT_DATA_LOST as an argument.

Note 2. When CPHA = 0 is set, wait for half a RSPCK cycle in software before resuming communication. For details, see “5.6 Resuming communication in slave mode and CPHA0”.

3.3 Configurations

For the SPI driver, configuration definitions that can be modified by the user are provided in the r_spi_cfg.h file.

3.3.1 Transmission Control Setting

This specifies the transmission control method.

Name: SPIn_TRANSMIT_CONTROL (n = 0, 1)

Table 3-1 Settings of SPIn_TRANSMIT_CONTROL

Setting	Description
SPI_USED_INTERRUPT (initial value)	Uses interrupts for transmission control.
SPI_USED_DMAC0	Uses DMAC0 for transmission control.
SPI_USED_DMAC1	Uses DMAC1 for transmission control.
SPI_USED_DMAC2	Uses DMAC2 for transmission control.
SPI_USED_DMAC3	Uses DMAC3 for transmission control.
SPI_USED_DTC	Uses DTC for transmission control

3.3.2 Reception Control Setting

This specifies the reception control method.

Name: SPIn_RECEIVE_CONTROL (n = 0, 1)

Table 3-2 Settings of SPIn_RECEIVE_CONTROL

Setting	Description
SPI_USED_INTERRUPT (initial value)	Uses interrupts for reception control.
SPI_USED_DMAC0	Uses DMAC0 for reception control.
SPI_USED_DMAC1	Uses DMAC1 for reception control.
SPI_USED_DMAC2	Uses DMAC2 for reception control.
SPI_USED_DMAC3	Uses DMAC3 for reception control.
SPI_USED_DTC	Uses DTC for reception control

3.3.3 SPTI Interrupt Priority Level

This specifies the priority level of the SPTIn interrupt. (n = 0, 1)

Name: SPIn_SPTI_PRIORITY

Table 3-3 Settings of SPIn_SPTI_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.

3.3.4 SPRI Interrupt Priority Level

This specifies the priority level of the SPRIn interrupt. (n = 0, 1)

Name: SPIn_SPRI_PRIORITY

Table 3-4 Settings of SPIn_SPRI_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.

3.3.5 SPII Interrupt Priority Level

This specifies the priority level of the SPIIn interrupt. (n = 0, 1)

Name: SPIn_SPII_PRIORITY

Table 3-5 Settings of SPIn_SPII_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.

3.3.6 SPEI Interrupt Priority Level

This specifies the priority level of the SPEIn interrupt. (n = 0, 1)

Name: SPIn_SPEI_PRIORITY

Table 3-6 Settings of SPIn_SPEI_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.

3.3.7 SPTEND Interrupt Priority Level

This specifies the priority level of the SPTENDn interrupt. (n = 0, 1)

Name: SPIn_SPTEND_PRIORITY

Table 3-7 Settings of SPIn_SPTEND_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.

3.3.8 Definition of Software-Controlled SSL Pin

This defines the SSL pin to be used with software control.

Name: SPIn_SS_PORT (Note), SPIn_SS_PIN (n = 0, 1)

Table 3-8 Settings of SPIn_SS_PORT and SPIn_SS_PIN

Name	Initial Value	Description
SPIn_SS_PORT (Note)	(PORT0->PODR)	Selects PORT0 as the SSL pin.
SPIn_SS_PIN	(0)	Selects PORTi00 as the SSL pin. (i is the port specified with SPIn_SS_PORT)

Note: By default, this definition is commented out.
When using the software-controlled SSL pin, un-comment this definition.

3.3.9 Function Allocation to RAM

This initializes the settings for executing specific functions of the SPI driver RAM.

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

Name: SPI_CFG_SECTION_xxx

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

Example: ARM_SPI_INITIALIZE function → SPI_CFG_SECTION_ARM_SPI_INITIALIZE

Table 3-9 Settings of SPI_CFG_SECTION_xxx

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

Table 3-10 Initial State of Function Allocation to RAM

No.	Function Name	Allocation to RAM
1	ARM_SPI_GetVersion	
2	ARM_SPI_GetCapabilities	
3	ARM_SPI_Initialize	
4	ARM_SPI_Uninitialize	
5	ARM_SPI_PowerControl	
6	ARM_SPI_Send	
7	ARM_SPI_Receive	
8	ARM_SPI_Transfer	
9	ARM_SPI_GetDataCount	
10	ARM_SPI_Control	
11	ARM_SPI_GetStatus	
12	spin_spti_interrupt (n = 0, 1) (SPTI interrupt handling process)	✓
13	spin_spri_interrupt (n = 0, 1) (SPRI interrupt handling process)	✓
14	spin_spii_interrupt (n = 0, 1) (SPII interrupt handling process)	✓
15	spin_spei_interrupt (n = 0, 1) (SPEI interrupt handling process)	✓
16	spin_sptend_interrupt (n = 0, 1) (SPTEND interrupt handling process)	✓

4. Detailed Information of Driver

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

4.1 Function Specifications

This section describes the specifications and processing flow of each function of the SPI driver.

The judgment, such as that for a conditional branch, is not always made as that described in the processing flow.

4.1.1 ARM_SPI_GetVersion Function

Table 4-1 ARM_SPI_GetVersion Function Specifications

Format	ARM_DRIVER_VERSION ARM_SPI_GetVersion(void)
Description	Acquires SPI driver version.
Argument	None
Return value	SPI driver version

Remarks	[Example of calling function from instance]
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	
	`main()`
	`{`
	`    ARM_DRIVER_VERSION version;`
	`    version = spi0Drv->GetVersion();`
	
	`}`

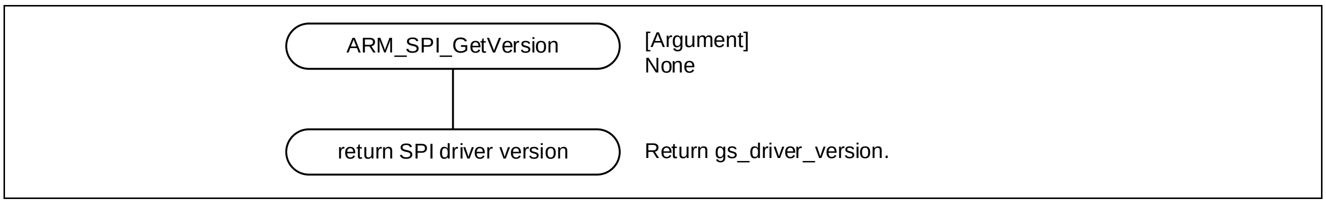


Figure 4-1 ARM_SPI_GetVersion Function Processing Flow

4.1.2 ARM_SPI_GetCapabilities Function

Table 4-2 ARM_SPI_GetCapabilities Function Specifications

Format	ARM_SPI_CAPABILITIES ARM_SPI_GetCapabilities(void)
Description	Acquires SPI driver functions.
Argument	None
Return value	Driver capabilities

Remarks	[Example of calling function from instance]
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	
	`main()`
	`{`
	`    ARM_SPI_CAPABILITIES cap;`
	`    cap = spi0Drv->GetCapabilities();`
	
	`}`

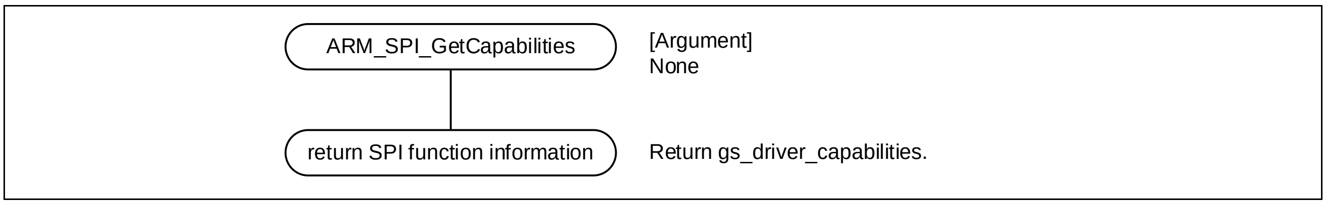

ARM_SPI_GetCapabilities
[Argument]
None
return SPI function information
Return gs_driver_capabilities.

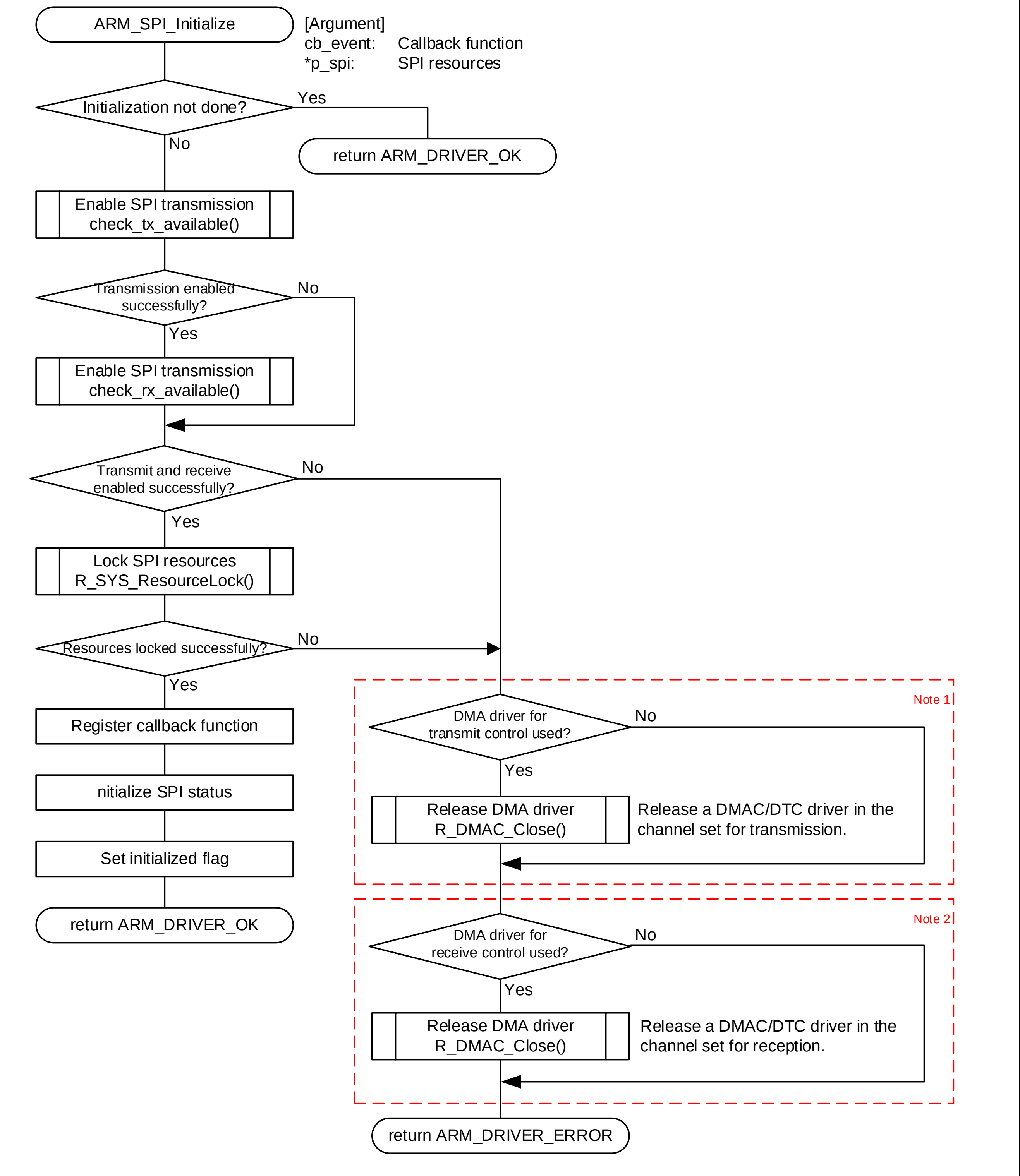
Figure 4-2 ARM_SPI_GetCapabilities Function Processing Flow

4.1.3 ARM_SPI_Initialize Function

Table 4-3 ARM_SPI_Initialize Function Specifications

Format	int32_t ARM_SPI_Initialize(ARM_SPI_SignalEvent_t cb_event, st_spi_resources_t * const p_spi)	
Description	Initializes the SPI driver (initializes RAM, makes register settings, and registers interrupts with NVIC).	
Argument	ARM_SPI_SignalEvent_t cb_event: Callback function Specifies the callback function to be executed when an event occurs. If NULL is set, the callback function will not be executed.	
	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources to be initialized.	
Return value	ARM_DRIVER_OK	SPI initialization completed
	ARM_DRIVER_ERROR	SPI initialization failed Initialization failure occurs if one of the following conditions is detected. • If neither transmission nor reception can be used (due to an erroneous setting in communication control, NVIC registration, etc.) • If the resources of the SPI channel to be used are locked (If SPin is already locked by the R_SYS_ResourceLock function)

Remarks	When this function is accessed, specifying the SPI resources is not required.
	
	[Example of calling function from instance]
	`static void callback(uint32_t event);`
	
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	
	`main()`
	`{`
	`    spi0Drv->Initialize(callback);`
	`}`	


ARM_SPI_Initialize
[Argument]
cb_event: Callback function
*p_spi: SPI resources
Initialization not done?
Yes
No
return ARM_DRIVER_OK
Enable SPI transmission check_tx_available()
Transmission enabled successfully?
No
Yes
Enable SPI transmission check_rx_available()
Transmit and receive enabled successfully?
No
Yes
Lock SPI resources R_SYS_ResourceLock()
Resources locked successfully?
No
Yes
Register callback function
nitialize SPI status
Set initialized flag
return ARM_DRIVER_OK
Note 1
DMA driver for transmit control used?
No
Yes
Release DMA driver R_DMAC_Close()
Release a DMAC/DTC driver in the channel set for transmission.
Note 2
DMA driver for receive control used?
No
Yes
Release DMA driver R_DMAC_Close()
Release a DMAC/DTC driver in the channel set for reception.
return ARM_DRIVER_ERROR

Note 1. When DMAC or DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DMAC or DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.

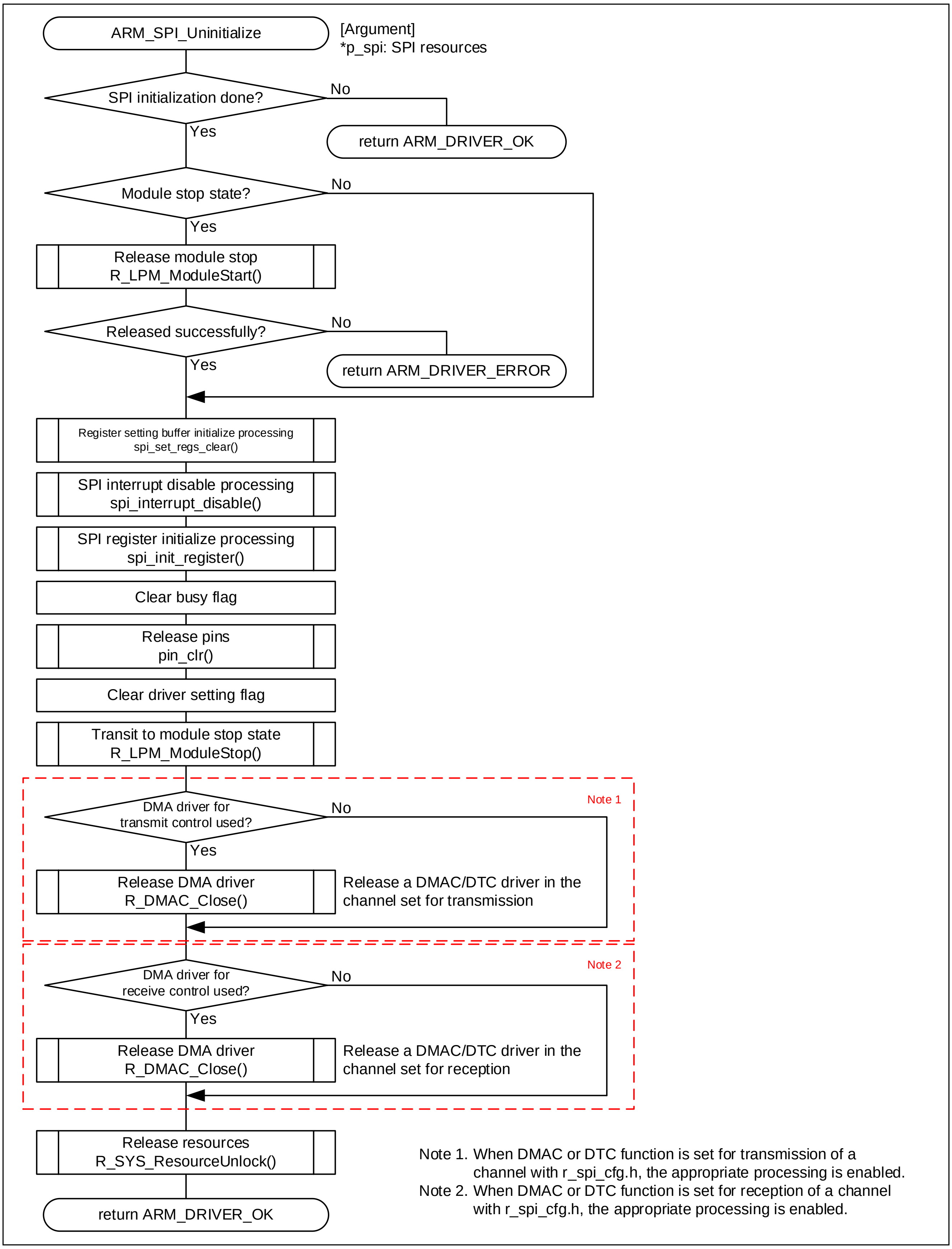
Figure 4-3 ARM_SPI_Initialize Function Processing Flow

4.1.4 ARM_SPI_Uninitialize Function

Table 4-4 ARM_SPI_Uninitialize Function Specifications

Format	int32_t ARM_SPI_Uninitialize(st_spi_resources_t * const p_spi)	
Description	Releases the SPI driver.	
Argument	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to be released.	
Return value	ARM_DRIVER_OK	SPI released normally
	ARM_DRIVER_ERROR	SPI release failed When executed while in both the power-off state and module start state (when an error has occurred upon R_LPM_ModuleStart), SPI release fails.

Remarks	When this function is accessed, specifying the SPI resources is not required.
	[Example of calling function from instance]
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	
	`main()`
	`{`
	`    spi0Drv->Uninitialize();`
	`}`	



Note 1. When DMAC or DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DMAC or DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-4 ARM_SPI_Uninitialize Function Processing Flow

4.1.5 ARM_SPI_PowerControl Function

Table 4-5 ARM_SPI_PowerControl Function Specifications

Format	int32_t ARM_SPI_PowerControl(ARM_POWER_STATE state, st_spi_resources_t * const p_spi)	
Description	Releases the SPI from the module stop state, or causes a transition to the state.	
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 SPI from the module stop state. ARM_POWER_LOW: This setting is not supported.	
	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to which power is supplied.	
Return value	ARM_DRIVER_OK	Power setting change completed
	ARM_DRIVER_ERROR	Power setting change failed Power setting change failure occurs if one of the following conditions is detected. • If this function is executed with the SPI uninitialized • If transition to the module stop state has failed (An error has occurred in R_LPM_ModuleStart.)
	ARM_DRIVER_ERROR_UNSUPPORTED	Specified power setting is not supported

Remarks	When this function is accessed, specifying the SPI resources is not required.
	
	[Example of calling function from instance]
	// SPI driver instance (SPI0)
	extern ARM_DRIVER_SPI Driver_SPI0;
	ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;
	
	main()
	{
	    spi0Drv-> PowerControl (ARM_POWER_FULL);
	}	

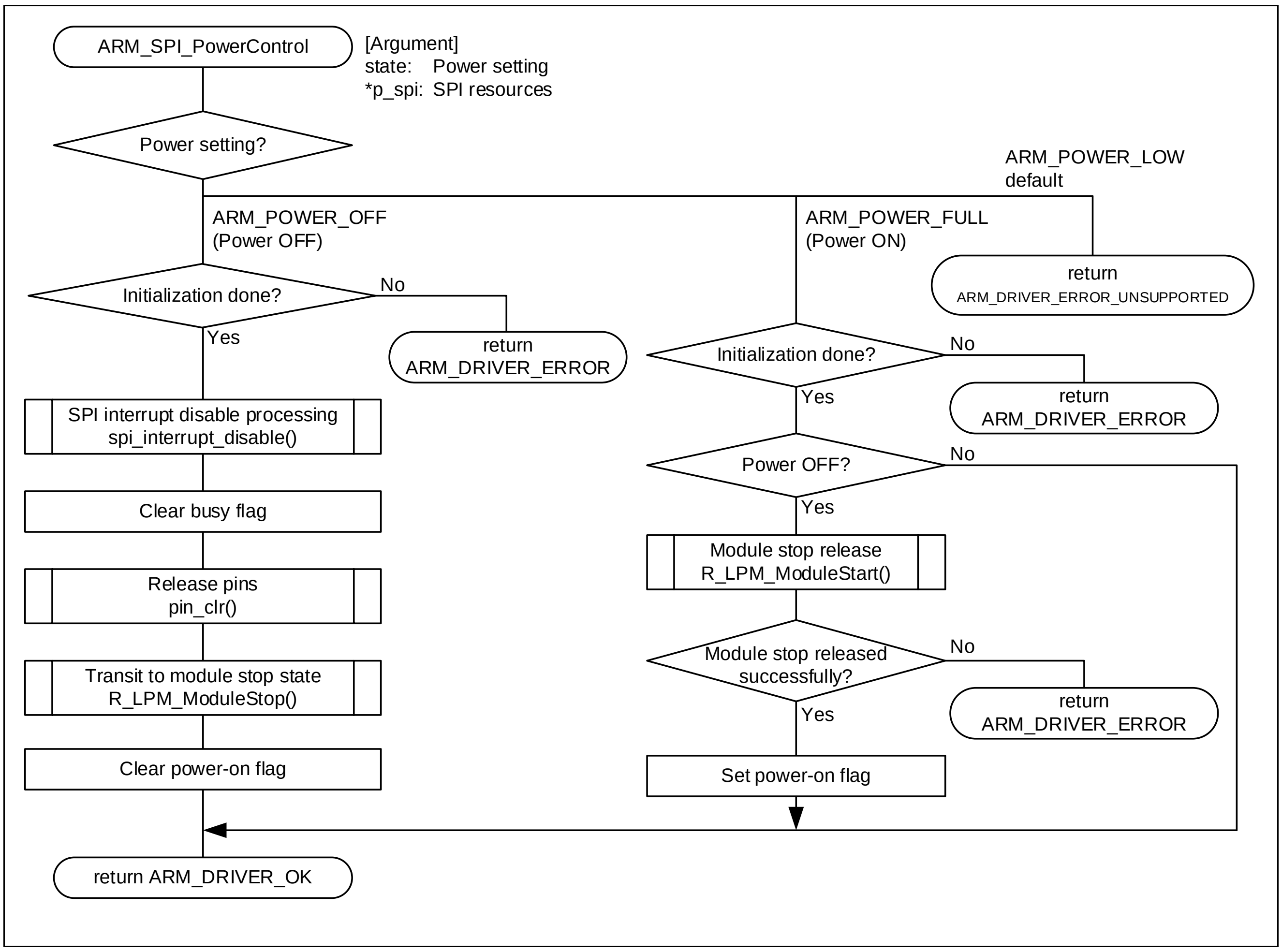


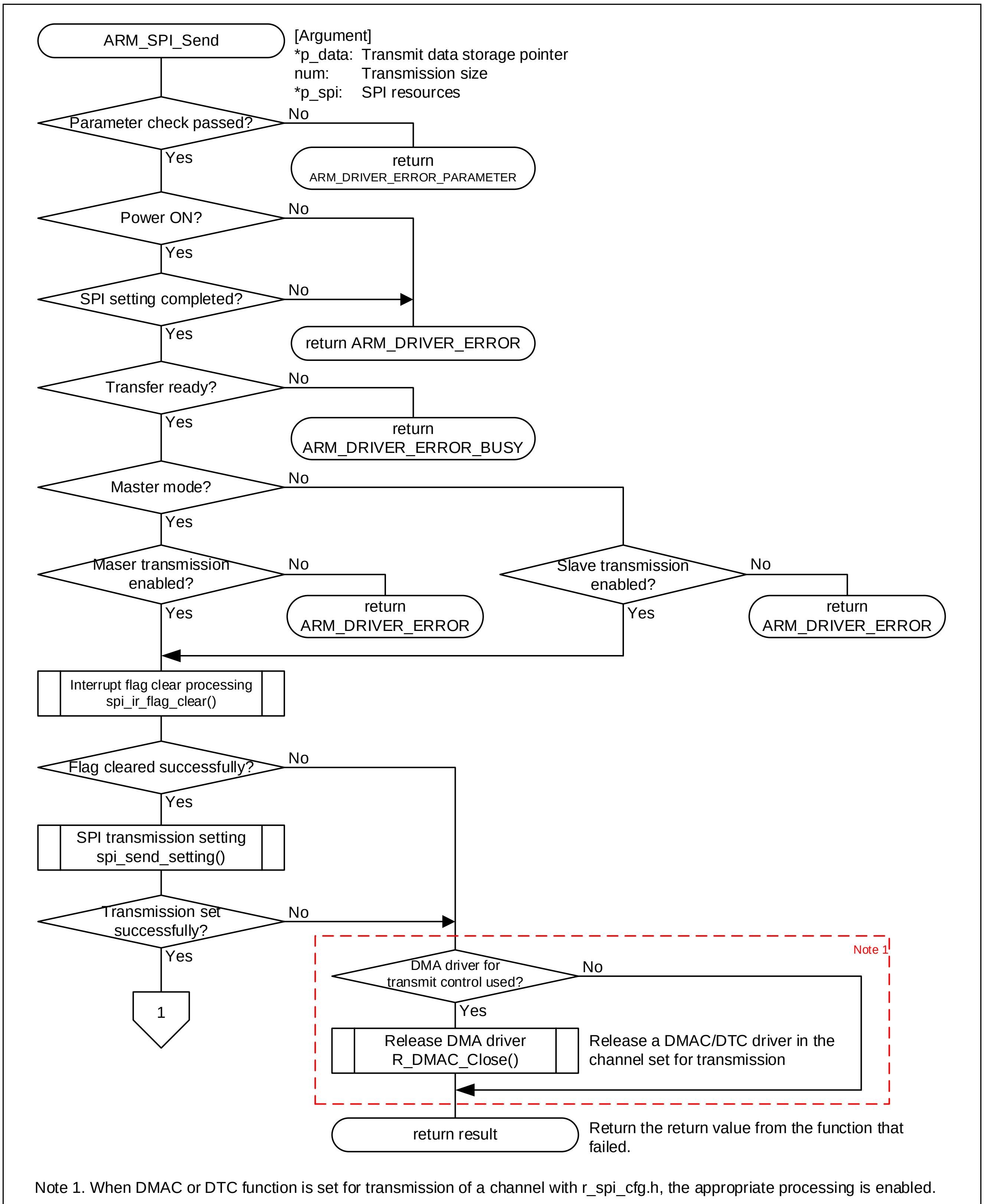
Figure 4-5 ARM_SPI_PowerControl Function Processing Flow

4.1.6 ARM_SPI_Send Function

Table 4-6 ARM_SPI_Send Function Specifications

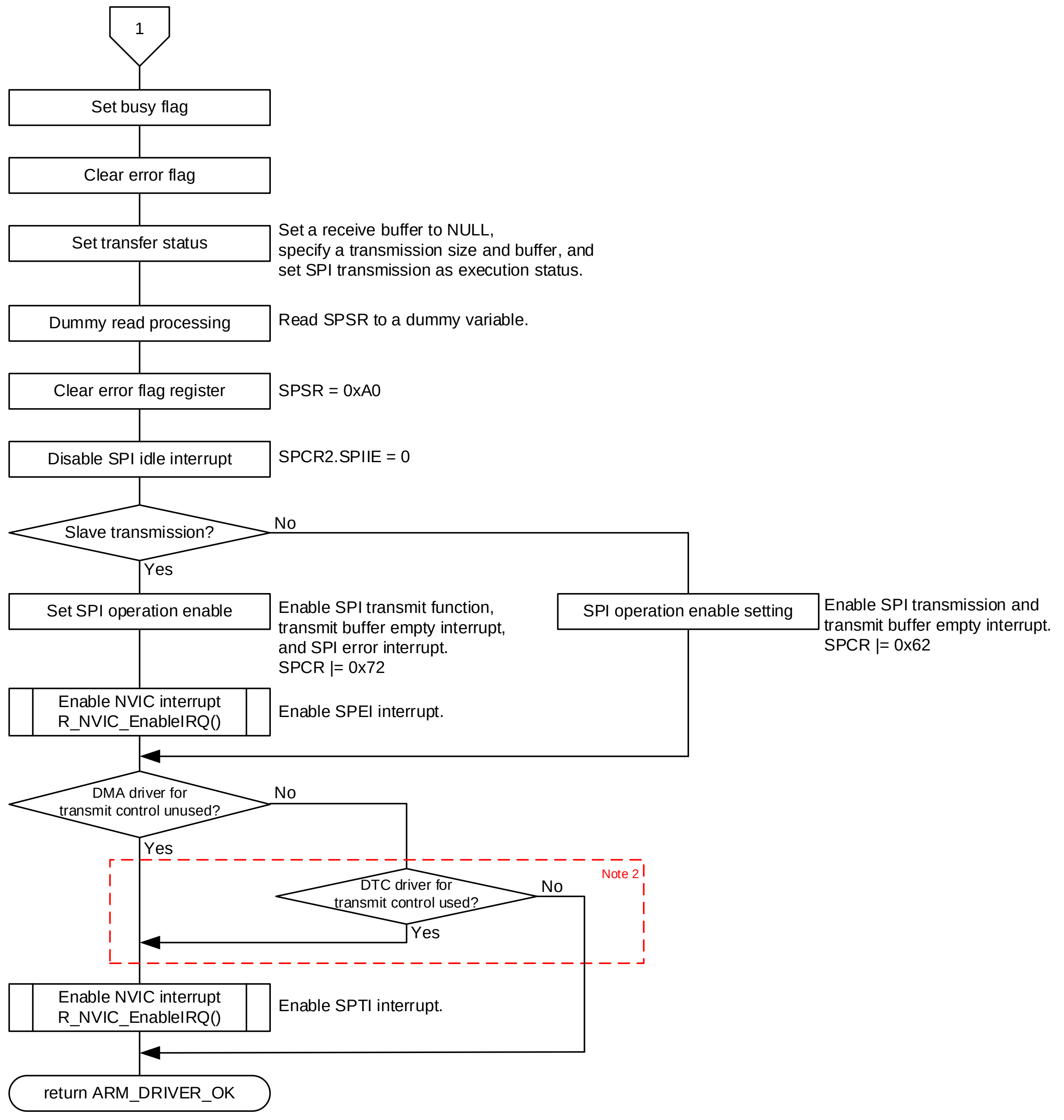
Format	int32_t ARM_SPI_Send(void const * const p_data, uint32_t num, st_spi_resources_t * const p_spi)	
Description	Starts transmission.	
Argument	void const * const *p_data : Transmit data storage pointer Specifies the start address of the buffer where data to be transmitted is stored.	
	uint32_t num : Transmission size Specifies the size of data to be transmitted.	
	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources that transmit data.	
Return value	ARM_DRIVER_OK	Transmission started normally
	ARM_DRIVER_ERROR Transmission start failure occurs if one of the following conditions is detected. • If this function is executed in the power OFF state • If this function is executed in the uninitialized state • If this function is executed with master mode set and moreover in master transmission disabled state • If this function is executed with slave mode set and moreover in slave transmission disabled state • If DMAC/DTC is specified for transmission and DMA driver setting has failed	Transmission start failed
	ARM_DRIVER_ERROR_BUSY When a communication-in-progress state is detected, transmission fails due to a busy state.	Transmission failed due to a busy state
	ARM_DRIVER_ERROR_PARAMETER A parameter error occurs if one of the following settings is specified. • The pointer to transmit data storage is set as NULL. • Transmit data size is set to 0.	Parameter error

Remarks	When this function is accessed, specifying the SPI resources is not required.
	
	[Example of calling function from instance]
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	`const uint8_t tx_data[2] = {0x51, 0xA2};`
	
	`main()`
	`{`
	`    spi0Drv->Send(&tx_data[0], 2);`
	`}`	


ARM_SPI_Send
[Argument]
*p_data: Transmit data storage pointer
num: Transmission size
*p_spi: SPI resources
Parameter check passed?
No
Yes
return ARM_DRIVER_ERROR_PARAMETER
Power ON?
No
Yes
SPI setting completed?
No
Yes
return ARM_DRIVER_ERROR
Transfer ready?
No
Yes
return ARM_DRIVER_ERROR_BUSY
Master mode?
No
Yes
Maser transmission enabled?
No
Yes
return ARM_DRIVER_ERROR
Slave transmission enabled?
No
Yes
return ARM_DRIVER_ERROR
Interrupt flag clear processing spi_ir_flag_clear()
Flag cleared successfully?
No
Yes
SPI transmission setting spi_send_setting()
Transmission set successfully?
No
Yes
1
Note 1
DMA driver for transmit control used?
No
Yes
Release DMA driver R_DMAC_Close()
Release a DMAC/DTC driver in the channel set for transmission
return result
Return the return value from the function that failed.

Note 1. When DMAC or DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-6 ARM_SPI_Send Function Processing Flow (1/2)



Note 2. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-7 ARM_SPI_Send Function Processing Flow (2/2)

4.1.7 ARM_SPI_Receive Function

Table 4-7 ARM_SPI_Receive Function Specifications

Format	int32_t ARM_SPI_Receive(void * const p_data, uint32_t num, st_spi_resources_t * const p_spi)	
Description	Starts reception.	
Argument	void * const p_data: Receive data storage pointer Specifies the start address of the buffer where received data is to be stored.	
	uint32_t num : Reception size Specifies the size of data to be received.	
	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources that receive data.	
Return value	ARM_DRIVER_OK	Reception started normally
	ARM_DRIVER_ERROR Reception start failure occurs if one of the following conditions is detected. • If this function is executed in the power OFF state • If this function is executed in the uninitialized state • If this function is executed with master mode set and moreover in master reception disabled state • If this function is executed with slave mode set and moreover in slave reception disabled state • If DMAC/DTC is specified for transmission and reception and DMA driver setting has failed	Reception start failed
	ARM_DRIVER_ERROR_BUSY If a communication-in-progress state is detected, reception fails due to a busy state.	Reception failed due to a busy state
	ARM_DRIVER_ERROR_PARAMETER A parameter error occurs if one of the following settings is specified. • The pointer to receive data storage is set as NULL. • Receive data size is set to 0.	Parameter error

Remarks	When this function is accessed, specifying the SPI resources is not required.
	
	[Example of calling function from instance]
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	`uint8_t rx_data[2];`
	
	`main()`
	`{`
	`    spi0Drv->Receive(&rx_data[0], 2);`
	`}`	

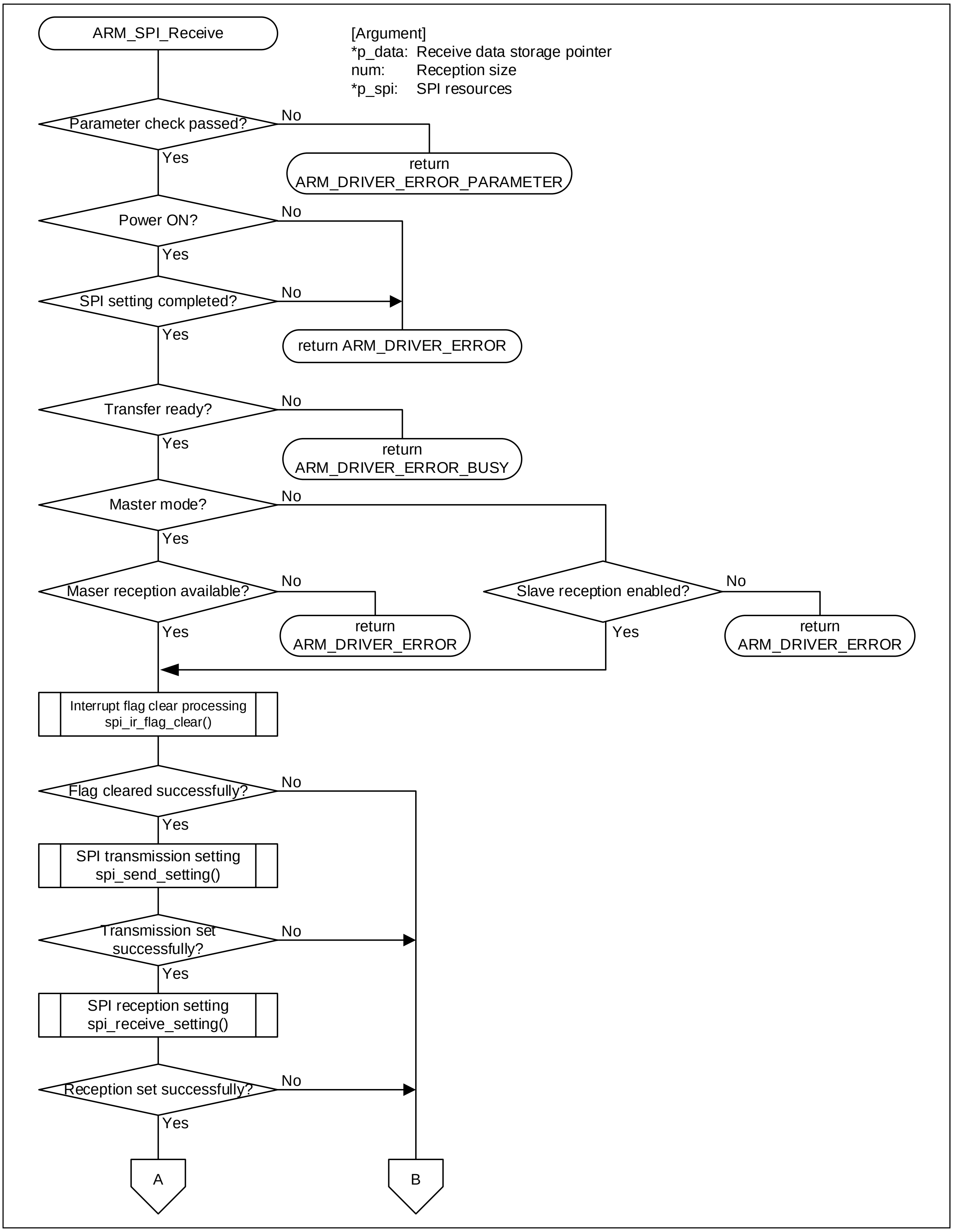
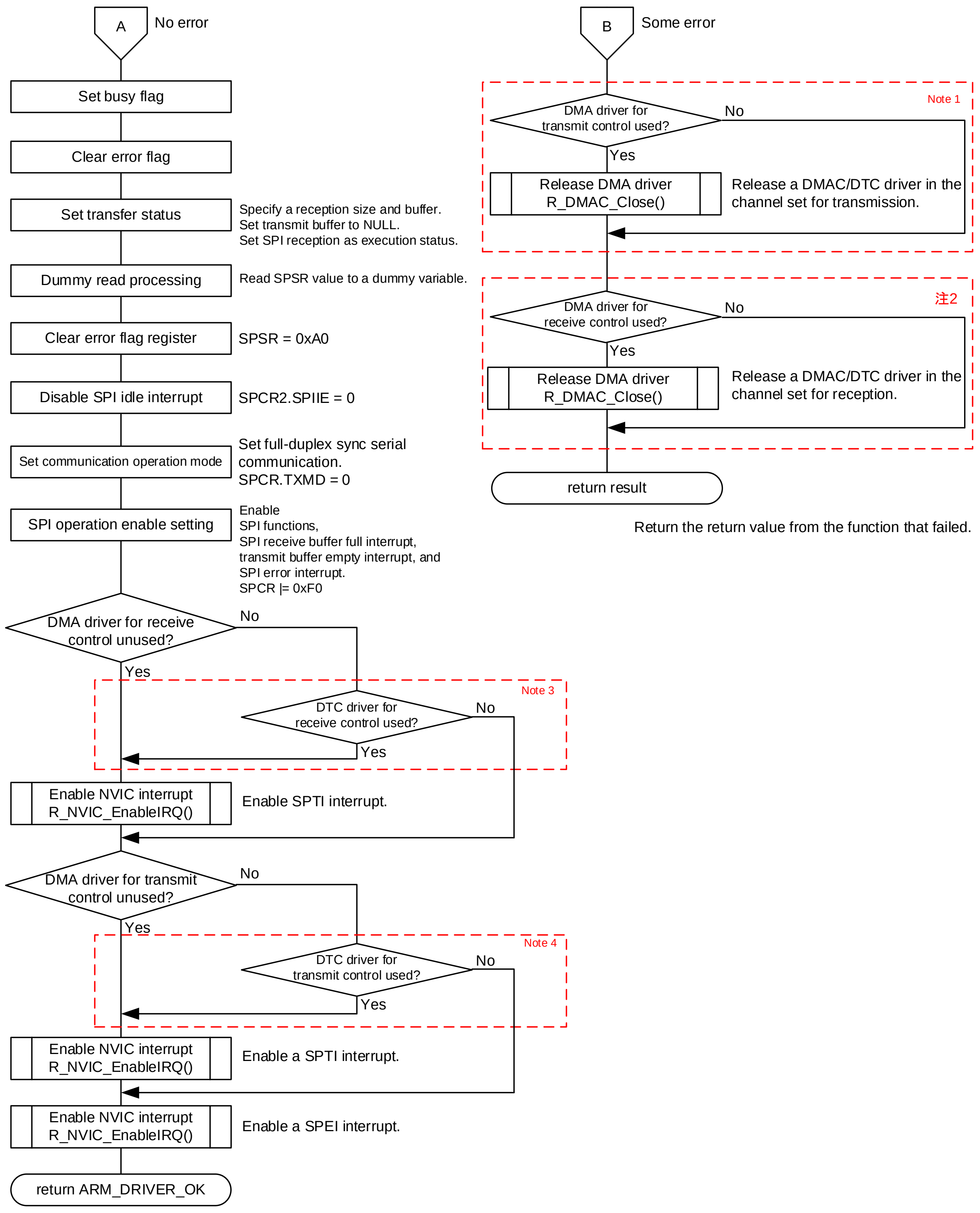

ARM_SPI_Receive
[Argument]
*p_data: Receive data storage pointer
num: Reception size
*p_spi: SPI resources
Parameter check passed?
No
Yes
return ARM_DRIVER_ERROR_PARAMETER
Power ON?
No
Yes
SPI setting completed?
No
Yes
return ARM_DRIVER_ERROR
Transfer ready?
No
Yes
return ARM_DRIVER_ERROR_BUSY
Master mode?
No
Yes
Maser reception available?
No
Yes
return ARM_DRIVER_ERROR
Slave reception enabled?
No
Yes
return ARM_DRIVER_ERROR
Interrupt flag clear processing spi_ir_flag_clear()
Flag cleared successfully?
No
Yes
SPI transmission setting spi_send_setting()
Transmission set successfully?
No
Yes
SPI reception setting spi_receive_setting()
Reception set successfully?
No
Yes
A
B

Figure 4-8 ARM_SPI_Receive Function Processing Flow (1/2)


A
No error
Set busy flag
Clear error flag
Set transfer status
Specify a reception size and buffer.
Set transmit buffer to NULL.
Set SPI reception as execution status.
Dummy read processing
Read SPSR value to a dummy variable.
Clear error flag register
SPSR = 0xA0
Disable SPI idle interrupt
SPCR2.SPIIE = 0
Set communication operation mode
Set full-duplex sync serial communication.
SPCR.TXMD = 0
SPI operation enable setting
Enable
SPI functions,
SPI receive buffer full interrupt,
transmit buffer empty interrupt, and
SPI error interrupt.
SPCR |= 0xF0
DMA driver for receive control unused?
No
Yes
Note 3
DTC driver for receive control used?
No
Yes
Enable NVIC interrupt R_NVIC_EnableIRQ()
Enable SPTI interrupt.
DMA driver for transmit control unused?
No
Yes
Note 4
DTC driver for transmit control used?
No
Yes
Enable NVIC interrupt R_NVIC_EnableIRQ()
Enable a SPTI interrupt.
Enable NVIC interrupt R_NVIC_EnableIRQ()
Enable a SPEI interrupt.
return ARM_DRIVER_OK
B
Some error
Note 1
DMA driver for transmit control used?
No
Yes
Release DMA driver R_DMAC_Close()
Release a DMAC/DTC driver in the channel set for transmission.
注2
DMA driver for receive control used?
No
Yes
Release DMA driver R_DMAC_Close()
Release a DMAC/DTC driver in the channel set for reception.
return result
Return the return value from the function that failed.

Note 1. When DMAC/DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DMAC/DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 3. When DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 4. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-9 ARM_SPI_Receive Function Processing Flow (2/2)

4.1.8 ARM_SPI_Transfer Function

Table 4-8 ARM_SPI_Transfer Function Specifications

Format	int32_t ARM_SPI_Transfer(void const * const p_data_out, void * const p_data_in, uint32_t num, st_spi_resources_t * const p_spi)
Description	Starts transmission and reception.
Argument	void const * const p_data_out : Transmit data storage pointer Specifies the start address of the buffer where data to be transmitted is stored.
	void * const p_data_in : Receive data storage pointer Specifies the start address of the buffer where received data is to be stored.
	uint32_t num : Transmission/reception size Specifies the size of data to be transmitted and received.
	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources that receive data.
Return value	ARM_DRIVER_OK Transmission/reception started normally
	ARM_DRIVER_ERROR Transmission/reception start failed Transmission/reception start failure occurs if one of the following conditions is detected. • If the function is executed in the power OFF state • If the function is executed in the uninitialized state • If this function is executed with master mode set and moreover in master reception disabled state • If this function is executed with slave mode set and moreover in slave reception disabled state • f DMAC/DTC is specified for transmission and reception and DMA driver setting has failed
	ARM_DRIVER_ERROR_BUSY Transmission/Reception failed due to a busy state If a communication-in-progress state is detected, transmission/reception fails due to a busy state.
	ARM_DRIVER_ERROR_PARAMETER Parameter error A parameter error occurs if one of the following conditions is detected. • If transmission/reception size is set to 0 • If the pointer to transmit data storage is set as NULL • If the pointer to receive data storage is set as NULL

Remarks	When this function is accessed, specifying the SPI resources is not required.
	
	[Example of calling function from instance]
	// SPI driver instance (SPI0)
	extern ARM_DRIVER_SPI Driver_SPI0;
	ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;
	uint8_t rx_data[2];
	const uint8_t tx_data[2] = {0x51, 0xA2};
	
	main()
	{
	    spi0Drv->Transfer (&tx_data[0], &rx_data[0], 2);
	}

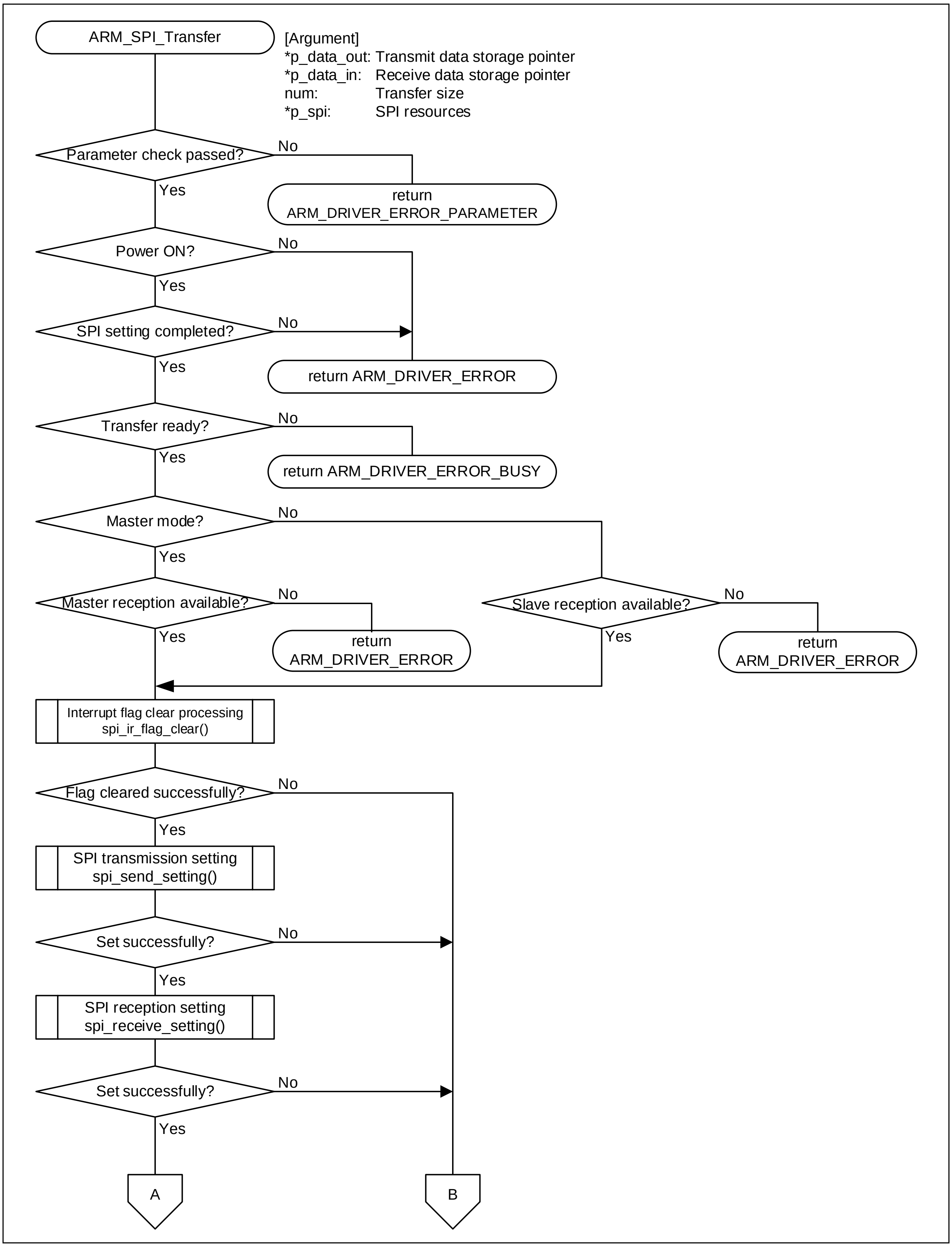
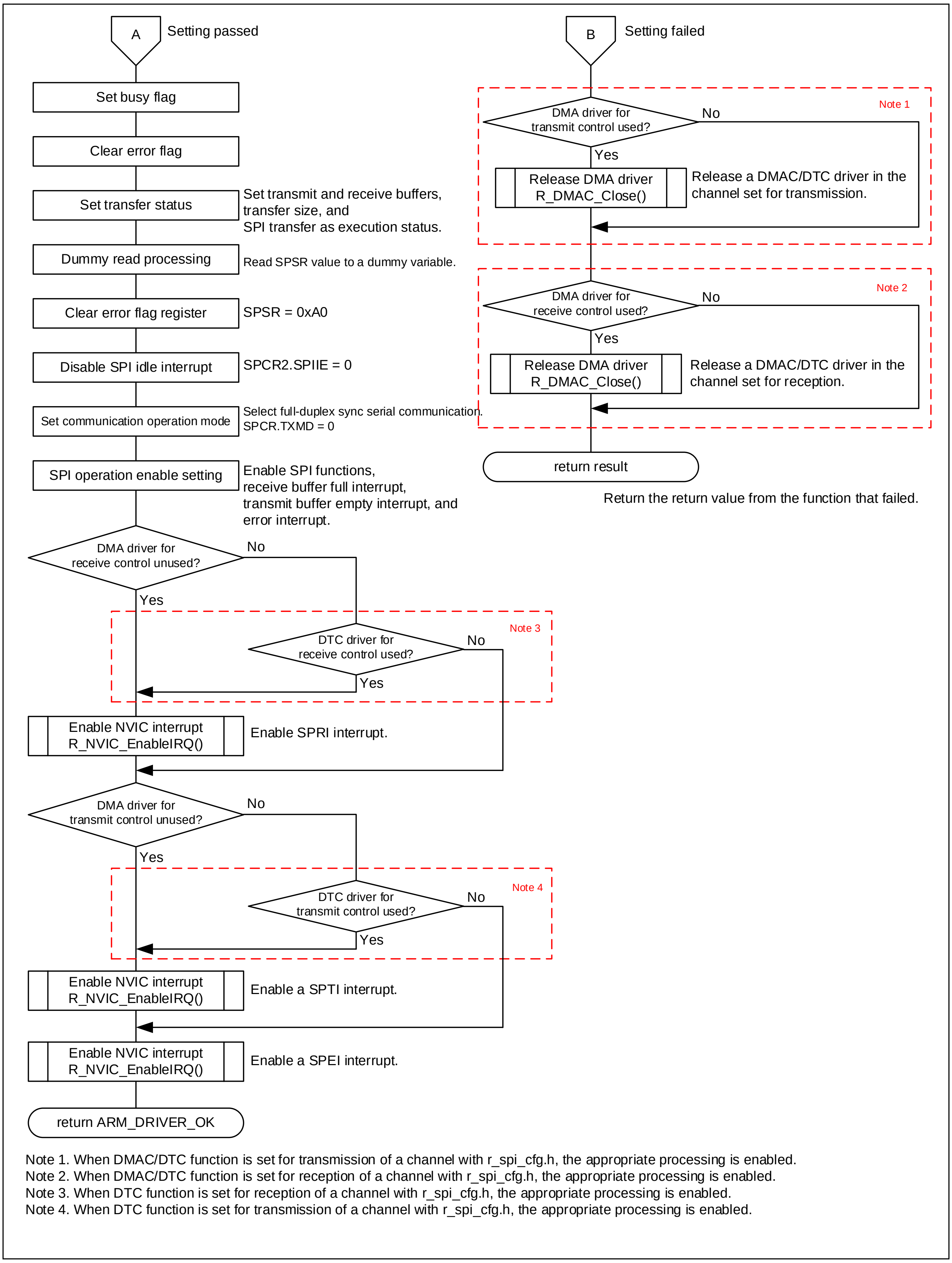

ARM_SPI_Transfer
[Argument]
*p_data_out: Transmit data storage pointer
*p_data_in: Receive data storage pointer
num: Transfer size
*p_spi: SPI resources
Parameter check passed?
No
return ARM_DRIVER_ERROR_PARAMETER
Yes
Power ON?
No
Yes
SPI setting completed?
No
Yes
return ARM_DRIVER_ERROR
Transfer ready?
No
return ARM_DRIVER_ERROR_BUSY
Yes
Master mode?
No
Yes
Master reception available?
No
return ARM_DRIVER_ERROR
Yes
Slave reception available?
No
return ARM_DRIVER_ERROR
Yes
Interrupt flag clear processing spi_ir_flag_clear()
Flag cleared successfully?
No
Yes
SPI transmission setting spi_send_setting()
Set successfully?
No
Yes
SPI reception setting spi_receive_setting()
Set successfully?
No
Yes
A
B

Figure 4-10 ARM_SPI_Transfer Function Processing Flow (1/2)


A
Setting passed
Set busy flag
Clear error flag
Set transfer status
Set transmit and receive buffers, transfer size, and SPI transfer as execution status.
Dummy read processing
Read SPSR value to a dummy variable.
Clear error flag register
SPSR = 0xA0
Disable SPI idle interrupt
SPCR2.SPIIE = 0
Set communication operation mode
Select full-duplex sync serial communication. SPCR.TXMD = 0
SPI operation enable setting
Enable SPI functions, receive buffer full interrupt, transmit buffer empty interrupt, and error interrupt.
DMA driver for receive control unused?
No
Yes
Note 3
DTC driver for receive control used?
No
Yes
Enable NVIC interrupt R_NVIC_EnableIRQ()
Enable SPRI interrupt.
DMA driver for transmit control unused?
No
Yes
Note 4
DTC driver for transmit control used?
No
Yes
Enable NVIC interrupt R_NVIC_EnableIRQ()
Enable a SPTI interrupt.
Enable NVIC interrupt R_NVIC_EnableIRQ()
Enable a SPEI interrupt.
return ARM_DRIVER_OK
B
Setting failed
Note 1
DMA driver for transmit control used?
No
Yes
Release DMA driver R_DMAC_Close()
Release a DMAC/DTC driver in the channel set for transmission.
Note 2
DMA driver for receive control used?
No
Yes
Release DMA driver R_DMAC_Close()
Release a DMAC/DTC driver in the channel set for reception.
return result
Return the return value from the function that failed.

Note 1. When DMAC/DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DMAC/DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 3. When DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 4. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-11 ARM_SPI_Transfer Function Processing Flow (2/2)

4.1.9 ARM_SPI_GetDataCount Function

Table 4-9 ARM_SPI_GetDataCount Function Specifications

Format	uint32_t ARM_SPI_GetDataCount(st_spi_resources_t const * const p_spi)
Description	Acquires the current transmission/reception count. When executed during a transmit operation, returns the transmitted data count; when executed during a receive operation or a transmit/receive operation, returns the received data count.
Argument	st_spi_resources_t * const p_spi : the SPI resources Specifies the resources of the SPI that get the transmission/reception count.
Return value	Transmission/reception count

Remarks	When this function is accessed, specifying the SPI resources is not required.
	
	[Example of calling function from instance]
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	
	`main()`
	`{`
	`    uint32_t tx_count;`
	`    tx_count = spi0Drv->GetDataCount();`
	`}`

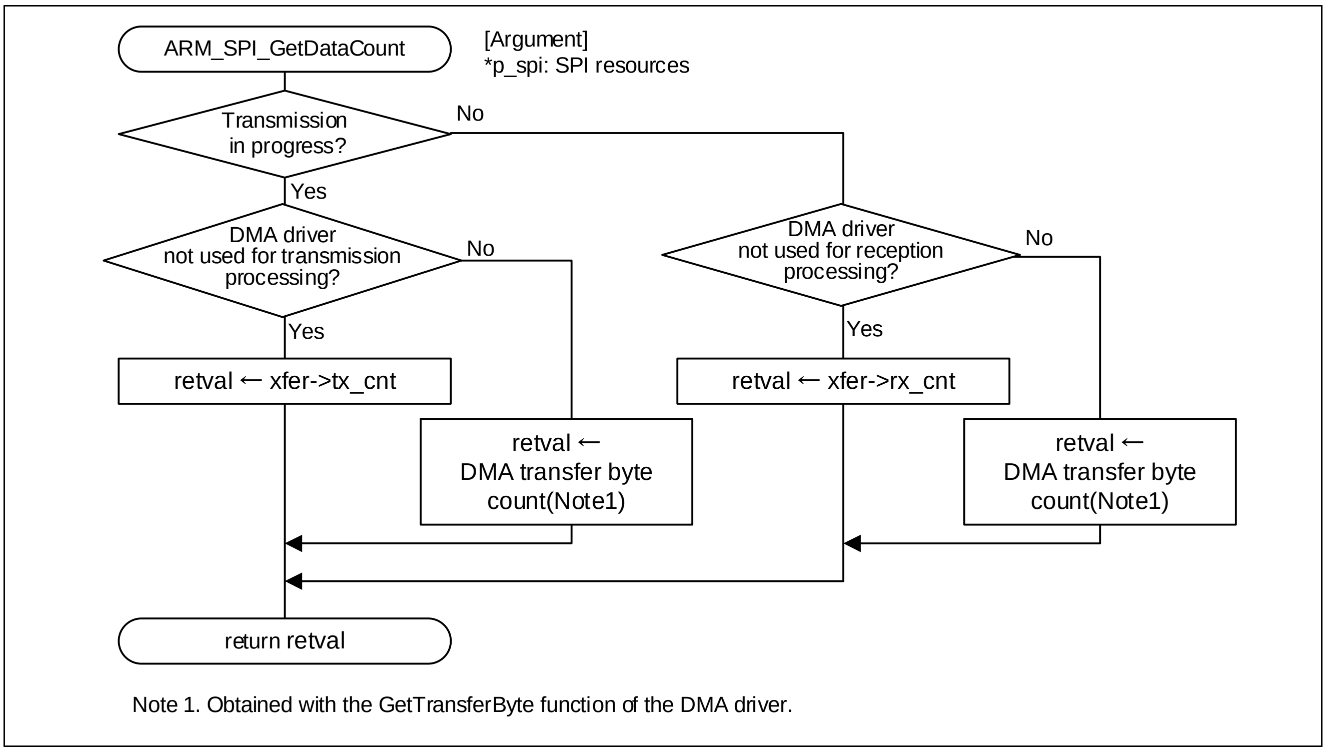

ARM_SPI_GetDataCount
[Argument]
*p_spi: SPI resources
Transmission in progress?
No
Yes
DMA driver not used for transmission processing?
No
Yes
retval ← xfer->tx_cnt
retval ← DMA transfer byte count(Note1)
DMA driver not used for reception processing?
No
Yes
retval ← xfer->rx_cnt
retval ← DMA transfer byte count(Note1)
return retval
Note 1. Obtained with the GetTransferByte function of the DMA driver.

Figure 4-12 ARM_SPI_GetDataCount Function Processing Flow

4.1.10 ARM_SPI_Control Function

Table 4-10 ARM_SPI_Control Function Specifications (1/2)

Format	int32_t ARM_SPI_Control(uint32_t control, uint32_t arg, st_spi_resources_t const * const p_spi)
Description	Executes a control command of the SPI.
Argument	uint32_t control : Control command See section 2.4.1, SPI Control Command Definitions for the control commands.
	uint32_t arg : Command-specific argument (See Table 4-11 for the relationship between control commands and arguments.)
	st_spi_resources_t * const p_spi : SPI resources Specify the SPI resources to be controlled.
Return value	ARM_DRIVER_OK Control command execution completed
	ARM_DRIVER_ERROR Control command execution failed Control command execution failure occurs if one of the following conditions is detected. • If the function is executed in the power OFF state • If an illegal command is executed. • If communication is in progress when any command other than ARM_SPI_ABORT_TRANSFER is executed.
	ARM_DRIVER_ERROR_BUSY Control command execution failed due to a busy state If communication is in progress when any command other than ARM_SPI_ABORT_TRANSFER is executed, the control command execution will fail due to a busy state.
	ARM_SPI_ERROR_FRAME_FORMAT Frame format error A frame format error occurs if one of the following conditions is detected. • If both ARM_SPI_SS_SLAVE_SW and ARM_SPI_CPOL0_CPHA0 are specified • If both ARM_SPI_SS_SLAVE_SW and ARM_SPI_CPOL1_CPHA0 are specified • If ARM_SPI_TI_SSI or ARM_SPI_MICROWIRE is specified
	ARM_SPI_ERROR_BIT_ORDER Bit order error If an invalid value is set for the bit order setting, a bit order error occurs.
	ARM_SPI_ERROR_DATA_BITS Data bit error If a value other than 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, or 32 is set for data bit length, a data bit error occurs.
	ARM_SPI_ERROR_SS_MODE SSL pin control setting error An SSL pin control setting error occurs if one of the following conditions is detected. • If, upon execution of the ARM_SPI_MODE_MASTER command, a value other than ARM_SPI_SS_MASTER_UNUSED, ARM_SPI_SS_MASTER_SW, or ARM_SPI_SS_MASTER_HW_OUTPUT is set for the SSL control setting • If, upon execution of the ARM_SPI_MODE_SLAVE command, a value other than ARM_SPI_SS_SLAVE_HW or ARM_SPI_SS_SLAVE_SW is set for the SSL control setting

Remarks	When this function is accessed, specifying the SPI resources is not required.
	
	[Example of calling function from instance]
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	
	`main()`
	`{`
	`    spi0Drv->Control(ARM_SPI_ABORT_TRANSFER, NULL);`
	`}`

Table 4-11 Operations Specified with Control Commands and Command-Specific Arguments

Control command (control)	Command-Specific Argument (arg)	Description
ARM_SPI_MODE_INACTIVE	NULL(0)	No arguments are used.
ARM_SPI_MODE_MASTER	Baud rate (MAX: PCLKA/2 MIN:PCLKA/4096)	Initializes master mode with the specified baud rate.
ARM_SPI_MODE_SLAVE	NULL(0)	No arguments are used.
ARM_SPI_SET_BUS_SPEED	Baud rate (MAX: PCLKA/2 MIN:PCLKA/4096)	Sets transfer speed. Specify a baud rate for the second argument.
ARM_SPI_GET_BUS_SPEED	NULL(0)	No arguments are used.
ARM_SPI_SET_DEFAULT_TX_VALUE	Default data (0x00 to 0xFFFFFFFF) (Note)	Sets the transmit data (default data) to be output during reception. Specify the default data value for the second argument.
ARM_SPI_CONTROL_SS	ARM_SPI_SS_INACTIVE	Sets the SSL0 pin to the inactive (High) state
	ARM_SPI_SS_ACTIVE	Sets the SSL0 pin to the active (Low) state
ARM_SPI_ABORT_TRANSFER	NULL(0)	No arguments are used.

Note: The maximum size of default data depends on data bit length setting.

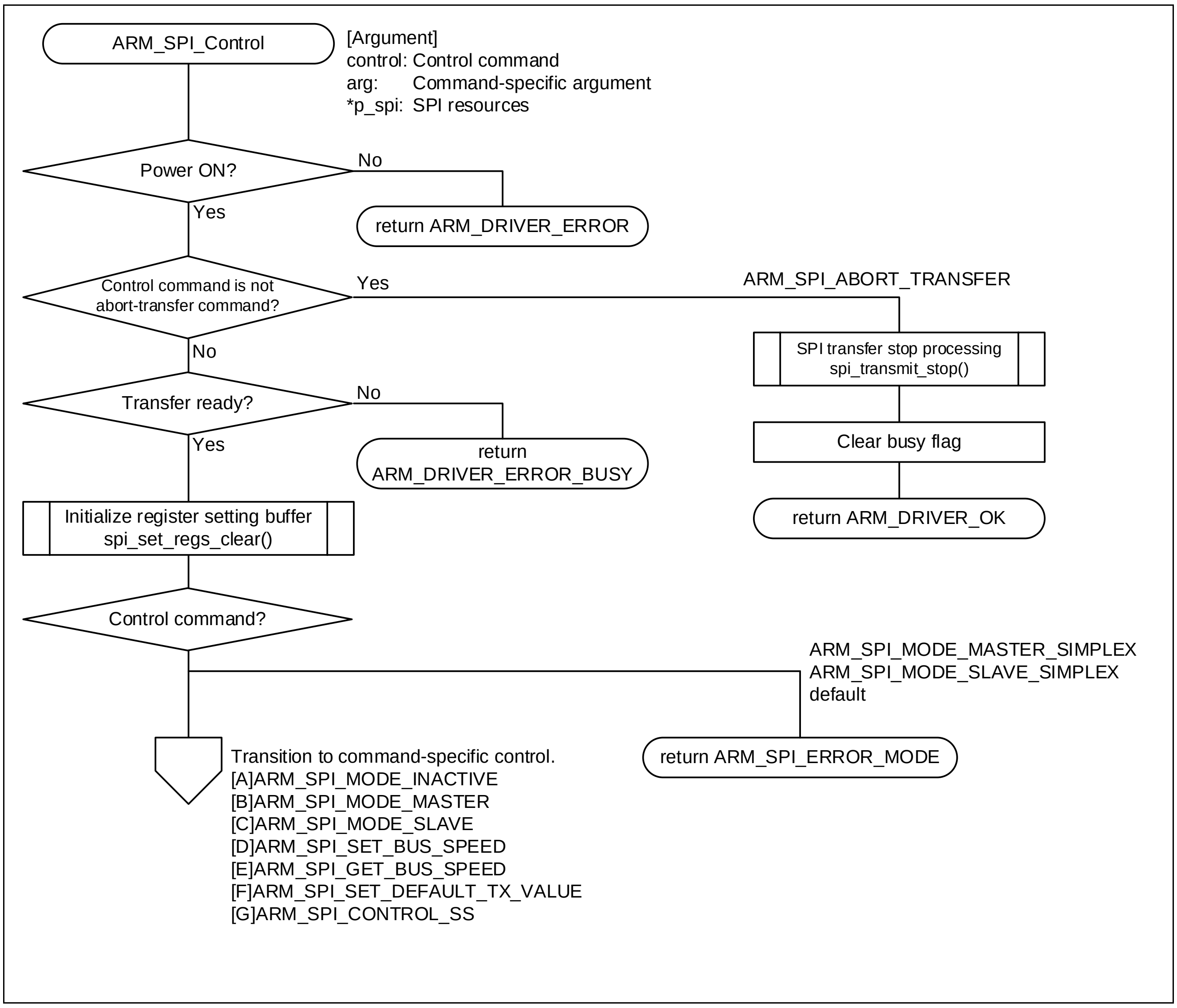

ARM_SPI_Control
[Argument]
control: Control command
arg: Command-specific argument
*p_spi: SPI resources
Power ON?
No
Yes
return ARM_DRIVER_ERROR
Control command is not abort-transfer command?
Yes
No
ARM_SPI_ABORT_TRANSFER
SPI transfer stop processing spi_transmit_stop()
Clear busy flag
return ARM_DRIVER_OK
Transfer ready?
No
Yes
return ARM_DRIVER_ERROR_BUSY
Initialize register setting buffer spi_set_regs_clear()
Control command?
ARM_SPI_MODE_MASTER_SIMPLEX
ARM_SPI_MODE_SLAVE_SIMPLEX
default
return ARM_SPI_ERROR_MODE
Transition to command-specific control.
[A]ARM_SPI_MODE_INACTIVE
[B]ARM_SPI_MODE_MASTER
[C]ARM_SPI_MODE_SLAVE
[D]ARM_SPI_SET_BUS_SPEED
[E]ARM_SPI_GET_BUS_SPEED
[F]ARM_SPI_SET_DEFAULT_TX_VALUE
[G]ARM_SPI_CONTROL_SS

Figure 4-13 ARM_SPI_Control Function Processing Flow (1/3)

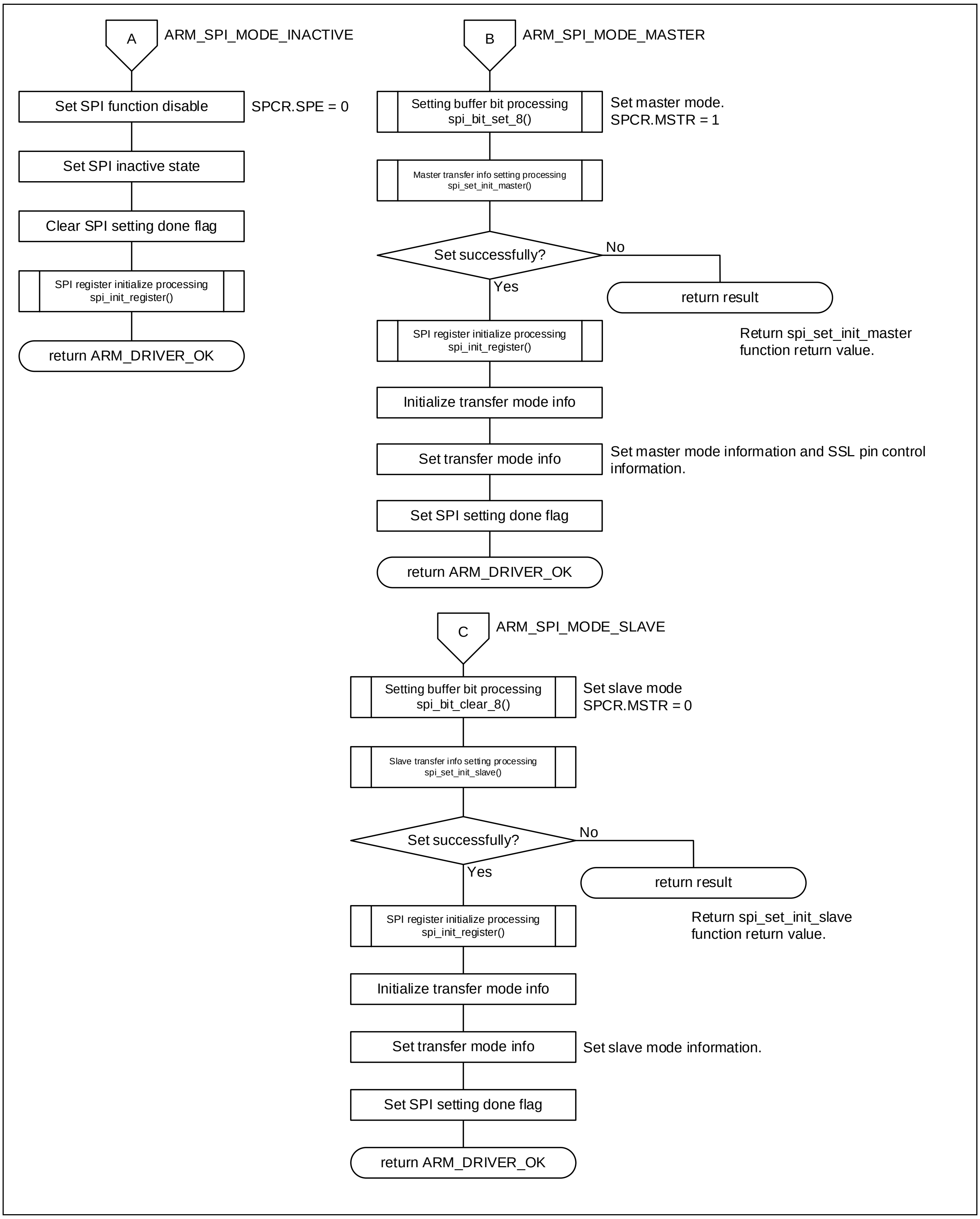

A
ARM_SPI_MODE_INACTIVE
Set SPI function disable
SPCR.SPE = 0
Set SPI inactive state
Clear SPI setting done flag
SPI register initialize processing spi_init_register()
return ARM_DRIVER_OK
B
ARM_SPI_MODE_MASTER
Setting buffer bit processing spi_bit_set_8()
Set master mode. SPCR.MSTR = 1
Master transfer info setting processing spi_set_init_master()
Set successfully?
No
Yes
return result
Return spi_set_init_master function return value.
SPI register initialize processing spi_init_register()
Initialize transfer mode info
Set transfer mode info
Set master mode information and SSL pin control information.
Set SPI setting done flag
return ARM_DRIVER_OK
C
ARM_SPI_MODE_SLAVE
Setting buffer bit processing spi_bit_clear_8()
Set slave mode SPCR.MSTR = 0
Slave transfer info setting processing spi_set_init_slave()
Set successfully?
No
Yes
return result
Return spi_set_init_slave function return value.
SPI register initialize processing spi_init_register()
Initialize transfer mode info
Set transfer mode info
Set slave mode information.
Set SPI setting done flag
return ARM_DRIVER_OK

Figure 4-14 ARM_SPI_Control Function Processing Flow (2/3)

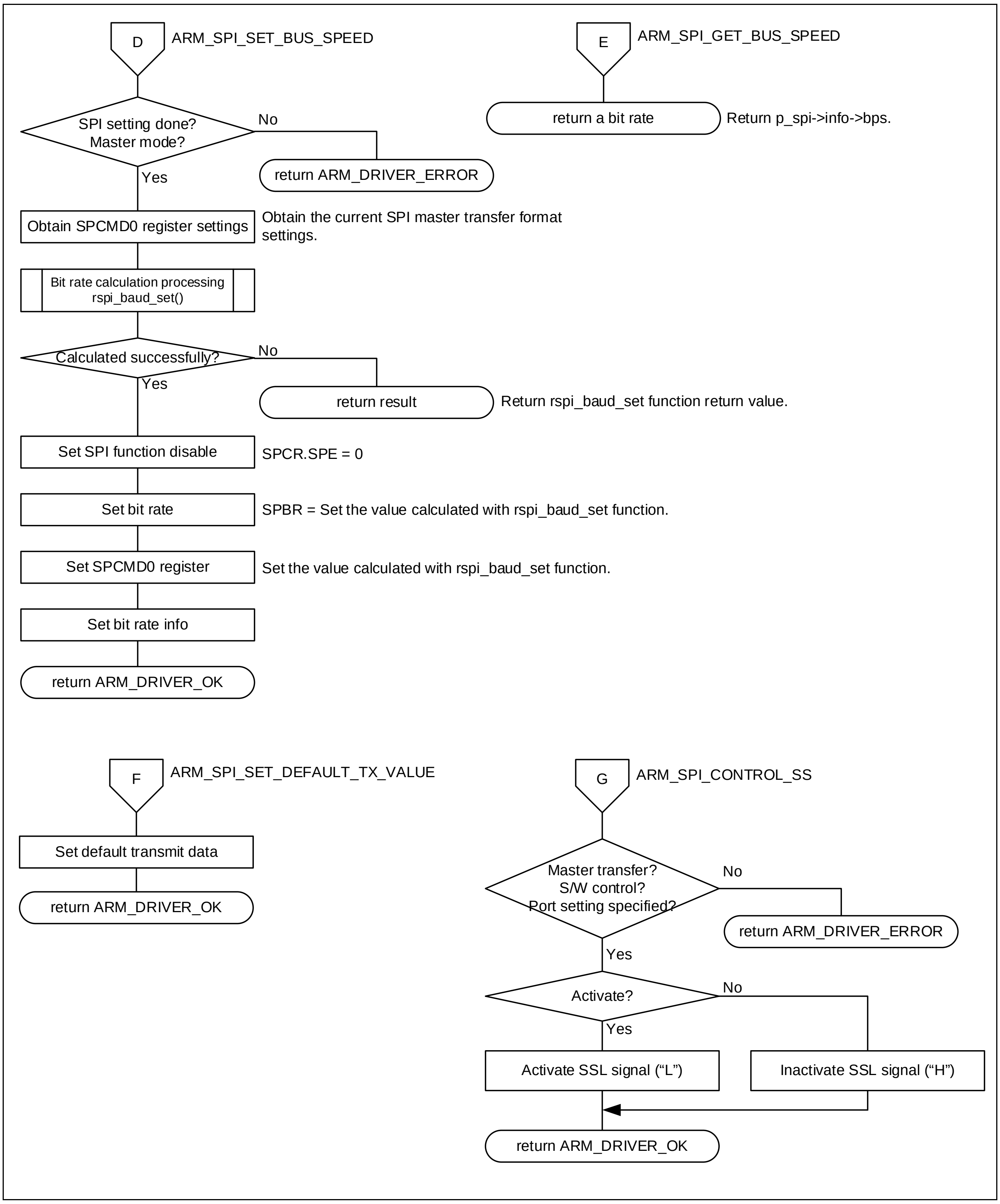

D
ARM_SPI_SET_BUS_SPEED
SPI setting done? Master mode?
No
return ARM_DRIVER_ERROR
Yes
Obtain SPCMD0 register settings
Obtain the current SPI master transfer format settings.
Bit rate calculation processing rspi_baud_set()
Calculated successfully?
No
return result
Return rspi_baud_set function return value.
Yes
Set SPI function disable
SPCR.SPE = 0
Set bit rate
SPBR = Set the value calculated with rspi_baud_set function.
Set SPCMD0 register
Set the value calculated with rspi_baud_set function.
Set bit rate info
return ARM_DRIVER_OK
E
ARM_SPI_GET_BUS_SPEED
return a bit rate
Return p_spi->info->bps.
F
ARM_SPI_SET_DEFAULT_TX_VALUE
Set default transmit data
return ARM_DRIVER_OK
G
ARM_SPI_CONTROL_SS
Master transfer? S/W control? Port setting specified?
No
return ARM_DRIVER_ERROR
Yes
Activate?
No
Yes
Activate SSL signal (“L”)
Inactivate SSL signal (“H”)
return ARM_DRIVER_OK

Figure 4-15 ARM_SPI_Control Function Processing Flow (3/3)

4.1.11 ARM_SPI_GetStatus Function

Table 4-12 ARM_SPI_GetStatus Function Specifications

Format	ARM_SPI_STATUS ARM_SPI_GetStatus(st_spi_resources_t const * const p_spi)
Description	Returns the status of the SPI.
Argument	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources concerned.
Return value	Communication status

Remarks	When this function is accessed, specifying the SPI resources is not required.
	
	[Example of calling function from instance]
	`// SPI driver instance ( SPI0 )`
	`extern ARM_DRIVER_SPI Driver_SPI0;`
	`ARM_DRIVER_SPI *spi0Drv = &Driver_SPI0;`
	
	`main()`
	`{`
	`    ARM_SPI_STATUS state;`
	`    state = spi0Drv->GetStatus();`
	
	`}`

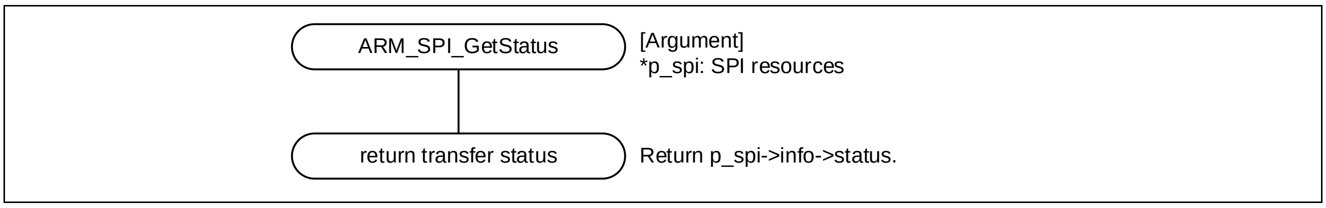


Figure 4-16 ARM_SPI_GetStatus Function Processing Flow

4.1.12 spi_set_init_master Function

Table 4-13 spi_set_init_master Function Specifications

Format	static int32_t spi_set_init_master(uint32_t control, uint32_t baudrate, st_spi_reg_buf_t * const p_spi_regs)	
Description	Specifies master mode settings.	
Argument	uint32_t control: Control command	
	uint32_t baudrate: Baud rate setting	
	st_spi_reg_set_t * const p_spi_regs: Pointer to register setting value storage	
Return value	ARM_DRIVER_OK	Master mode set normally
	ARM_DRIVER_ERROR	Master mode setting failed
	Master mode setting failure occurs if the baud rate setting is invalid (outside the range of a division ratio between 2 and 4096 of PCLKA).	
	ARM_SPI_ERROR_SS_MODE	SSL pin control setting error
	An SSL pin control setting error occurs if a value other than ARM_SPI_SS_MASTER_UNUSED, ARM_SPI_SS_MASTER_SW, or ARM_SPI_SS_MASTER_HW_OUTPUT is set for the SSL control setting.	
	ARM_SPI_ERROR_FRAME_FORMAT	Frame format error
	If ARM_SPI_TI_SSI or ARM_SPI_MICROWIRE is specified, a frame format error will occur.	
	ARM_SPI_ERROR_BIT_ORDER	Bit order error
	If an invalid value is set for the bit order setting, a bit order error occurs.	
	ARM_SPI_ERROR_DATA_BITS	Data bit error
	A data bit error occurs if a value other than 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, or 32 is specified for data bit length.	
Remarks		

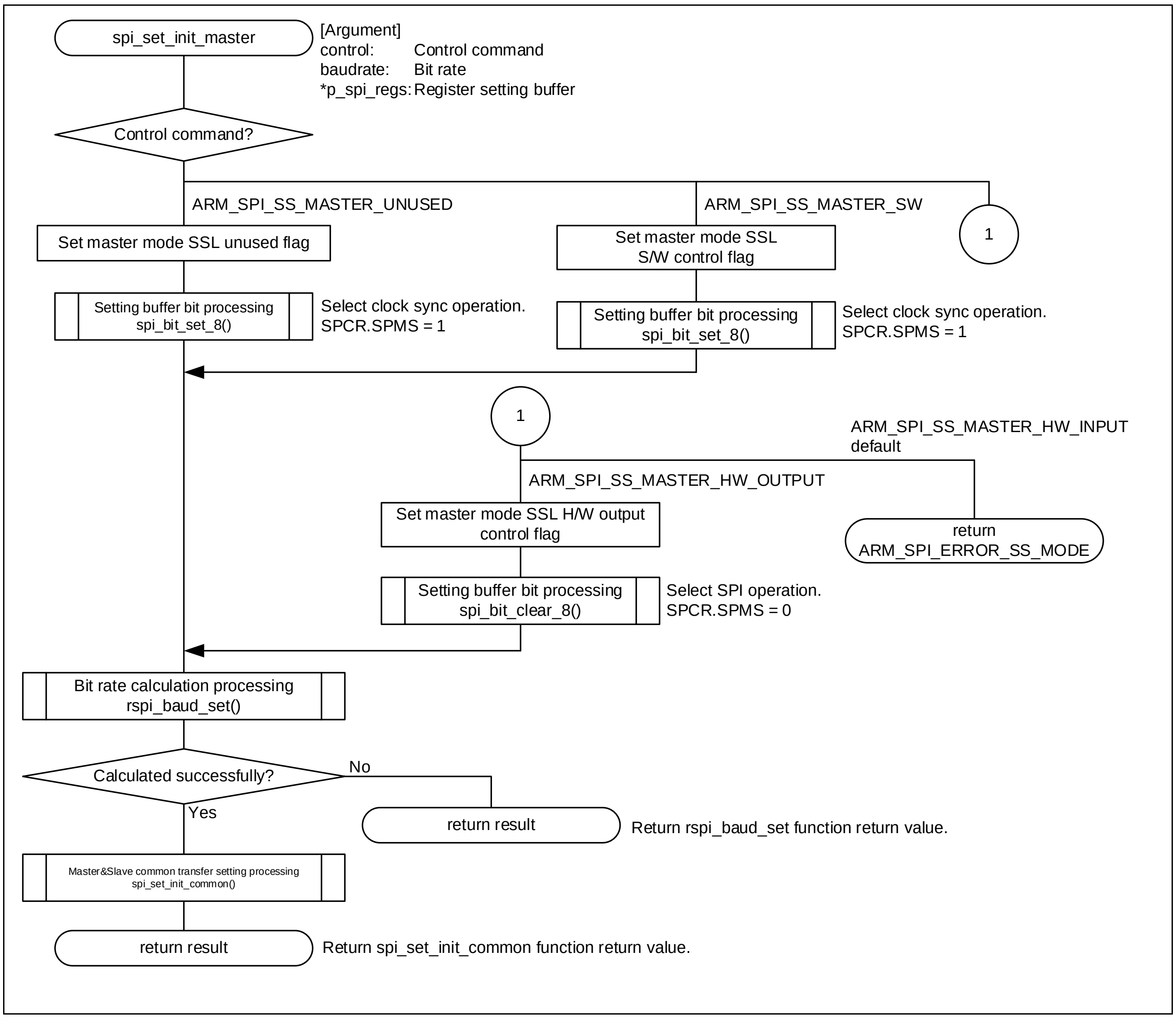

spi_set_init_master
[Argument]
control: Control command
baudrate: Bit rate
*p_spi_regs: Register setting buffer
Control command?
ARM_SPI_SS_MASTER_UNUSED
ARM_SPI_SS_MASTER_SW
Set master mode SSL unused flag
Set master mode SSL S/W control flag
1
Setting buffer bit processing spi_bit_set_8()
Select clock sync operation. SPCR.SPMS = 1
Setting buffer bit processing spi_bit_set_8()
Select clock sync operation. SPCR.SPMS = 1
1
ARM_SPI_SS_MASTER_HW_INPUT default
ARM_SPI_SS_MASTER_HW_OUTPUT
Set master mode SSL H/W output control flag
return ARM_SPI_ERROR_SS_MODE
Setting buffer bit processing spi_bit_clear_8()
Select SPI operation. SPCR.SPMS = 0
Bit rate calculation processing rspi_baud_set()
Calculated successfully?
No
Yes
return result
Return rspi_baud_set function return value.
Master&Slave common transfer setting processing spi_set_init_common()
return result
Return spi_set_init_common function return value.

Figure 4-17 spi_set_init_master Function Processing Flow

4.1.13 spi_set_init_slave Function

Table 4-14 spi_set_init_slave Function Specifications

Format	static int32_t spi_set_init_slave(uint32_t control, st_spi_reg_buf_t * const p_spi_regs)	
Description	Specifies slave mode settings.	
Argument	uint32_t control: Control command	
	st_spi_reg_set_t * const p_spi_regs: Pointer to register setting value storage	
Return value	ARM_DRIVER_OK	Slave mode set normally
	ARM_SPI_ERROR_SS_MODE	SSL pin control setting error
	An SSL pin control setting error occurs when a value other than ARM_SPI_SS_SLAVE_HW or ARM_SPI_SS_SLAVE_SW is set for the SSL control setting.	
Remarks		

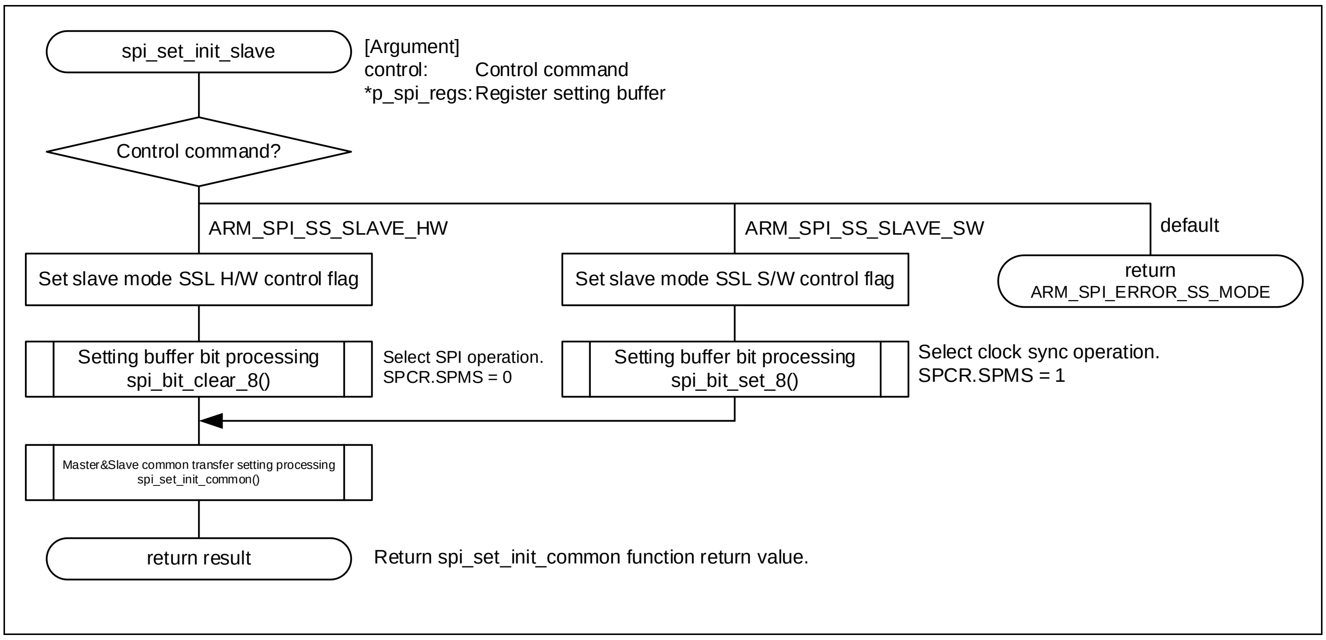

spi_set_init_slave
[Argument]
control: Control command
*p_spi_regs: Register setting buffer
Control command?
ARM_SPI_SS_SLAVE_HW
ARM_SPI_SS_SLAVE_SW
default
Set slave mode SSL H/W control flag
Set slave mode SSL S/W control flag
return ARM_SPI_ERROR_SS_MODE
Setting buffer bit processing spi_bit_clear_8()
Select SPI operation. SPCR.SPMS = 0
Setting buffer bit processing spi_bit_set_8()
Select clock sync operation. SPCR.SPMS = 1
Master&Slave common transfer setting processing spi_set_init_common()
return result
Return spi_set_init_common function return value.

Figure 4-18 spi_set_init_slave Function Processing Flow

4.1.14 spi_set_init_common Function

Table 4-15 spi_set_init_common Function Specifications

Format	static int32_t spi_set_init_common(uint32_t control, st_spi_reg_buf_t * const p_spi_regs)	
Description	Specifies transfer settings common to master and slave modes.	
Argument	uint32_t control: Control command	
	st_spi_reg_set_t * const p_spi_regs: Pointer to register setting value storage	
Return value	ARM_DRIVER_OK	The settings common to master and slave modes set successfully
	ARM_SPI_ERROR_FRAME_FORMAT A frame format error occurs if one of the following conditions is detected. • If both ARM_SPI_SS_SLAVE_SW and ARM_SPI_CPOL0_CPHA0 are specified • If both ARM_SPI_SS_SLAVE_SW and ARM_SPI_CPOL1_CPHA0 are specified • If ARM_SPI_TI_SSI or ARM_SPI_MICROWIRE is specified	Frame format error
	ARM_SPI_ERROR_BIT_ORDER If an invalid value is set for the bit order setting, a bit order error occurs.	Bit order error
	ARM_SPI_ERROR_DATA_BITS A data bit error occurs if a value other than 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, or 32 is specified for data bit length.	Data bit error
Remarks		

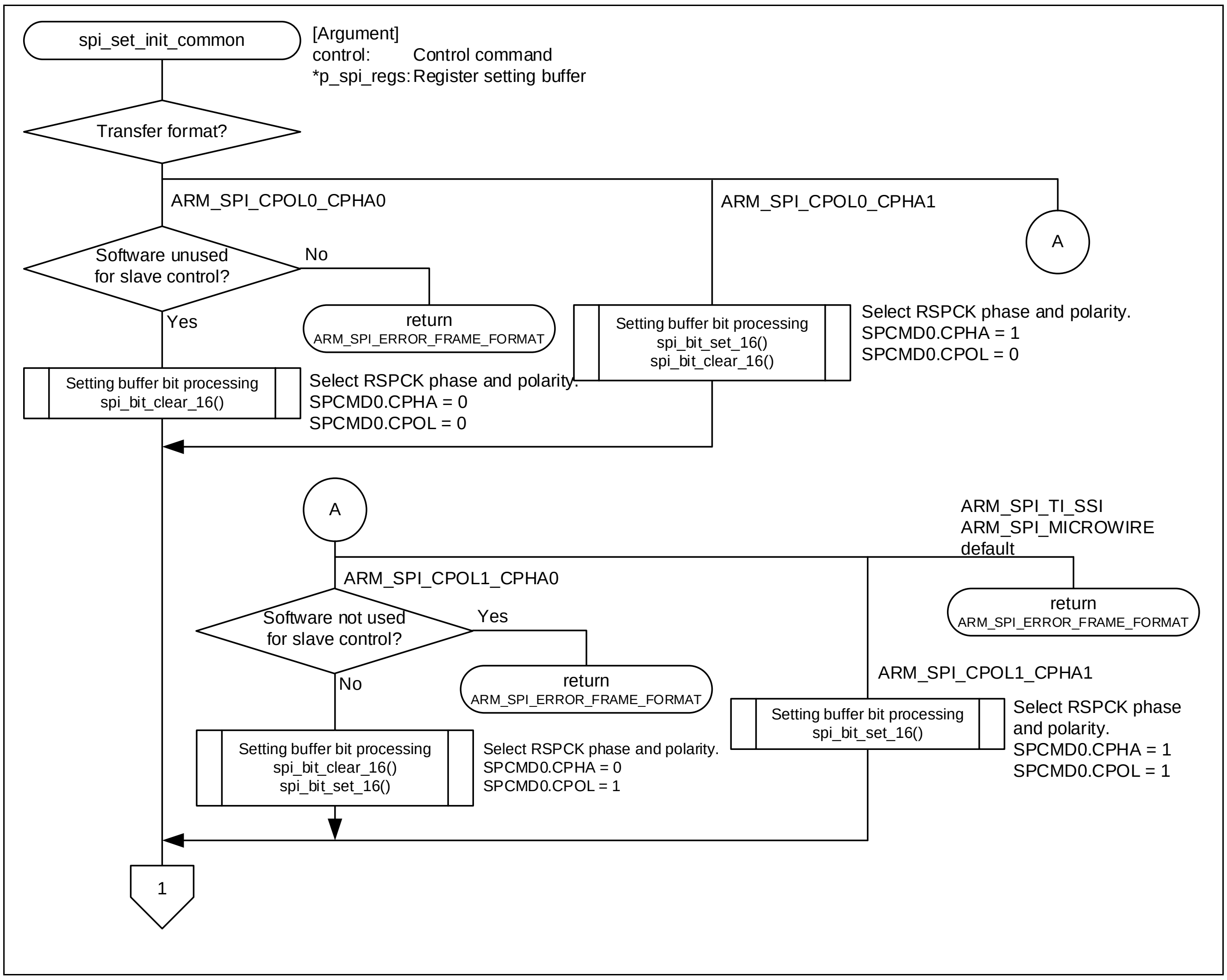

spi_set_init_common
[Argument]
control: Control command
*p_spi_regs: Register setting buffer
Transfer format?
ARM_SPI_CPOL0_CPHA0
ARM_SPI_CPOL0_CPHA1
A
Software unused for slave control?
No
Yes
return ARM_SPI_ERROR_FRAME_FORMAT
Setting buffer bit processing spi_bit_set_16() spi_bit_clear_16()
Select RSPCK phase and polarity. SPCMD0.CPHA = 1 SPCMD0.CPOL = 0
Setting buffer bit processing spi_bit_clear_16()
Select RSPCK phase and polarity. SPCMD0.CPHA = 0 SPCMD0.CPOL = 0
A
ARM_SPI_TI_SSI ARM_SPI_MICROWIRE default
ARM_SPI_CPOL1_CPHA0
return ARM_SPI_ERROR_FRAME_FORMAT
Software not used for slave control?
Yes
return ARM_SPI_ERROR_FRAME_FORMAT
No
ARM_SPI_CPOL1_CPHA1
Setting buffer bit processing spi_bit_set_16()
Select RSPCK phase and polarity. SPCMD0.CPHA = 1 SPCMD0.CPOL = 1
Setting buffer bit processing spi_bit_clear_16() spi_bit_set_16()
Select RSPCK phase and polarity. SPCMD0.CPHA = 0 SPCMD0.CPOL = 1
1

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

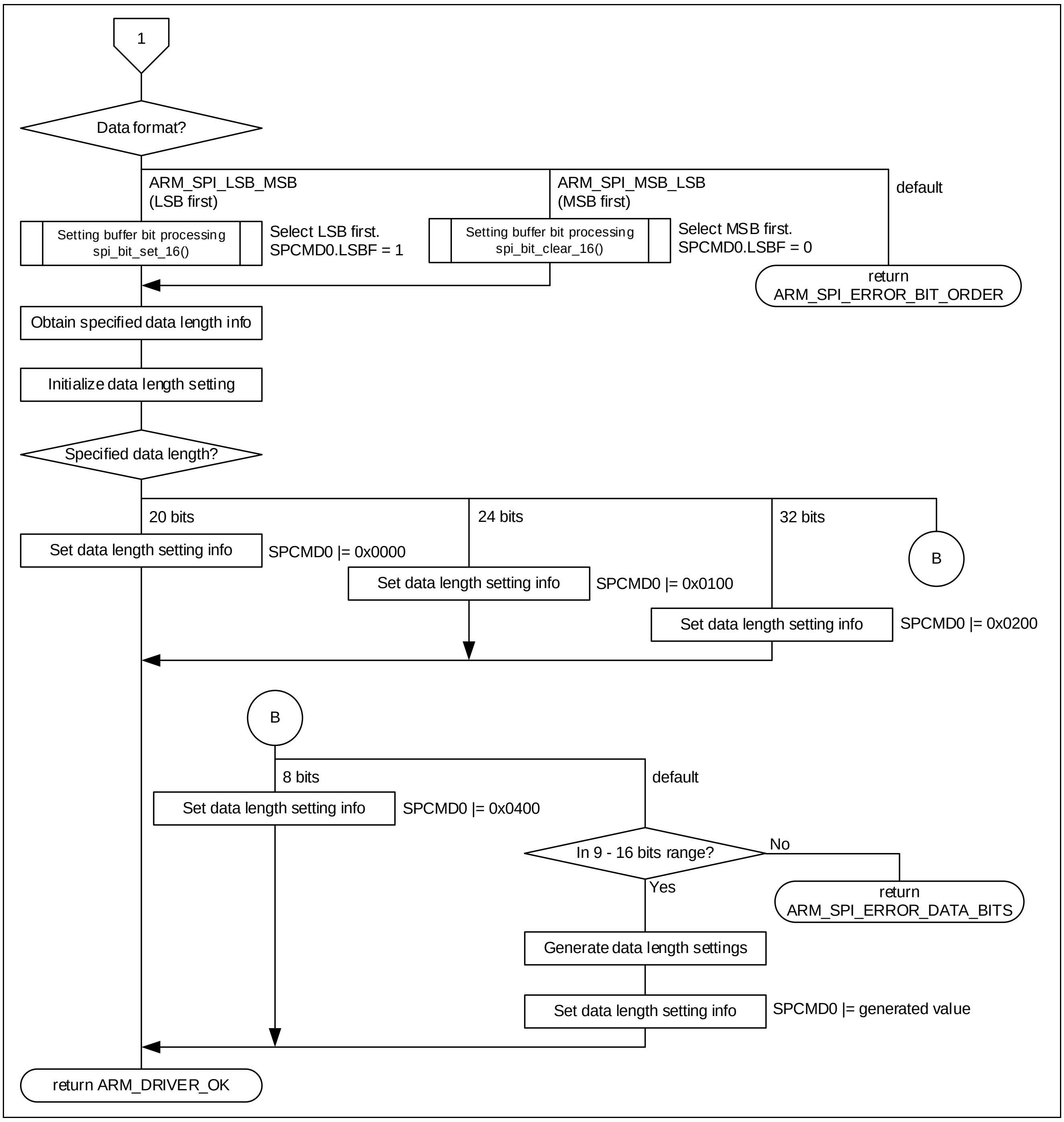


Figure 4-20 spi_set_init_common Function Processing Flow (2/2)

4.1.15 rspi_baud_set Function

Table 4-16 rspi_baud_set Function Specifications

Format	static int32_t rspi_baud_set(st_spi_reg_buf_t * const p_spi_regs, uint32_t bps_target)	
Description	Calculates a baud rate.	
Argument	st_spi_reg_buf_t * const p_spi_regs: Pointer to register setting buffer Pointer to buffer for storing baud rate calculation results	
	uint32_t bps_target: Baud rate settings	
Return value	ARM_DRIVER_OK	Baud rate calculation completed.
	ARM_DRIVER_ERROR A baud rate setting error occurs if the baud rate setting is invalid (outside the range of a division ratio between 2 and 4096 of PCLKA).	Baud rate calculation failed.
Remarks		

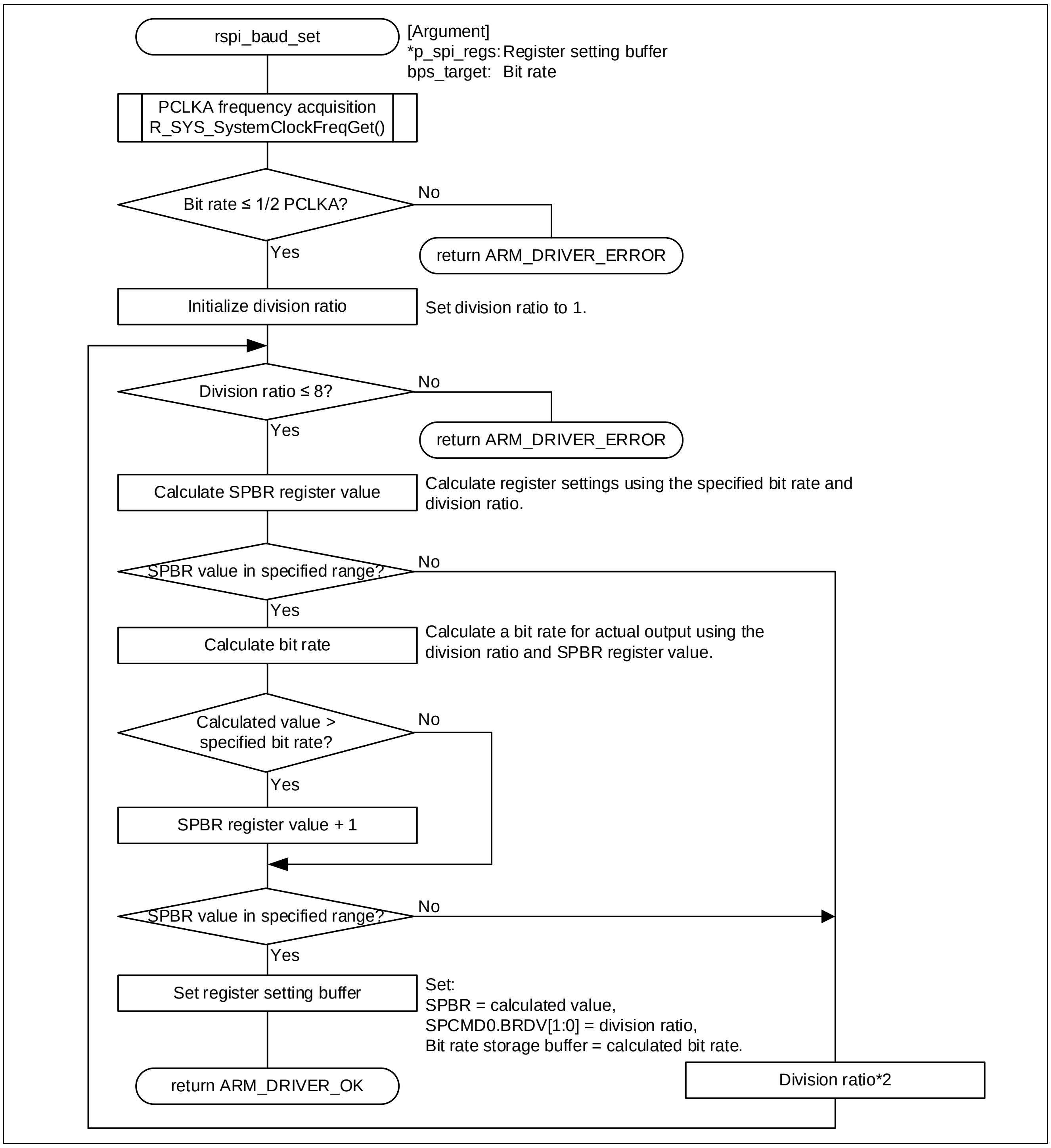

rspi_baud_set
[Argument]
*p_spi_regs: Register setting buffer
bps_target: Bit rate
PCLKA frequency acquisition R_SYS_SystemClockFreqGet()
Bit rate ≤ 1/2 PCLKA?
No
return ARM_DRIVER_ERROR
Yes
Initialize division ratio
Set division ratio to 1.
Division ratio ≤ 8?
No
return ARM_DRIVER_ERROR
Yes
Calculate SPBR register value
Calculate register settings using the specified bit rate and division ratio.
SPBR value in specified range?
No
Yes
Calculate bit rate
Calculate a bit rate for actual output using the division ratio and SPBR register value.
Calculated value > specified bit rate?
No
Yes
SPBR register value + 1
SPBR value in specified range?
No
Yes
Set register setting buffer
Set:
SPBR = calculated value,
SPCMD0.BRDV[1:0] = division ratio,
Bit rate storage buffer = calculated bit rate.
return ARM_DRIVER_OK
Division ratio*2

Figure 4-21 rspi_baud_set Function Processing Flow

4.1.16 spi_set_regs_clear Function

Table 4-17 spi_set_regs_clear Function Specifications

Format	static void spi_set_regs_clear(st_spi_reg_set_t * const p_regs)
Description	Initializes the register setting buffer.
Argument	st_spi_reg_buf_t * const p_spi_regs : Pointer to the register setting buffer
Return value	None
Remarks	–

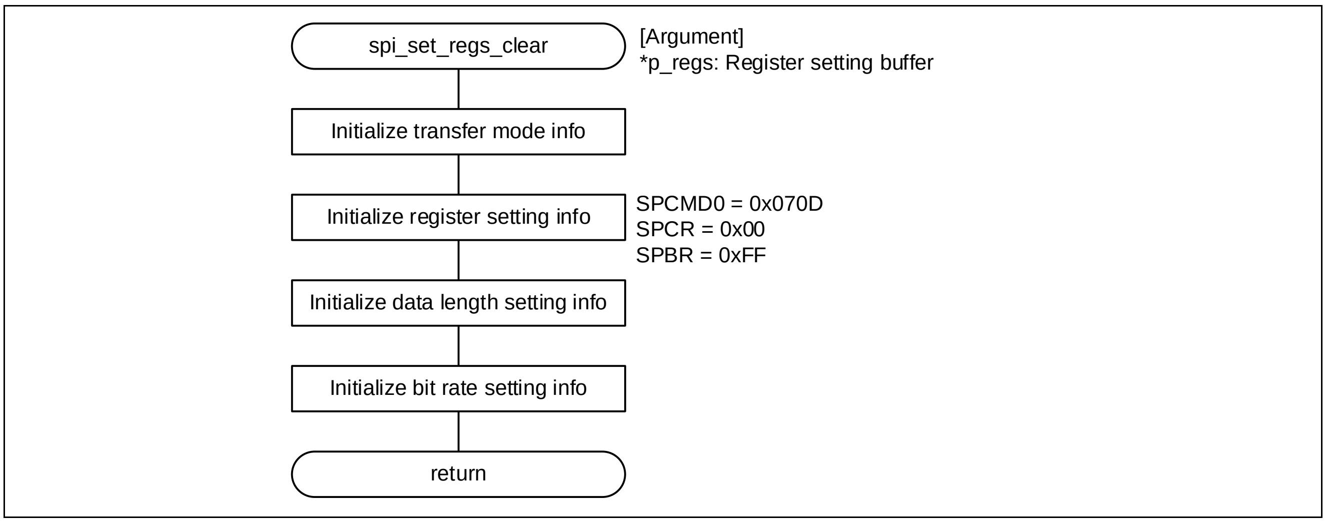

spi_set_regs_clear
[Argument]
*p_regs: Register setting buffer
Initialize transfer mode info
Initialize register setting info
SPCMD0 = 0x070D
SPCR = 0x00
SPBR = 0xFF
Initialize data length setting info
Initialize bit rate setting info
return

Figure 4-22 spi_set_regs_clear Function Processing Flow

4.1.17 spi_init_register Function

Table 4-18 spi_init_register Function Specifications

Format	static void spi_init_register(st_spi_reg_buf_t const * const p_spi_regs, st_spi_resources_t * const p_spi)
Description	Initializes the SPI capability setting registers.
Argument	st_spi_reg_buf_t * const p_spi_regs : Pointer to the register setting buffer
	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to be controlled.
Return value	None
Remarks	

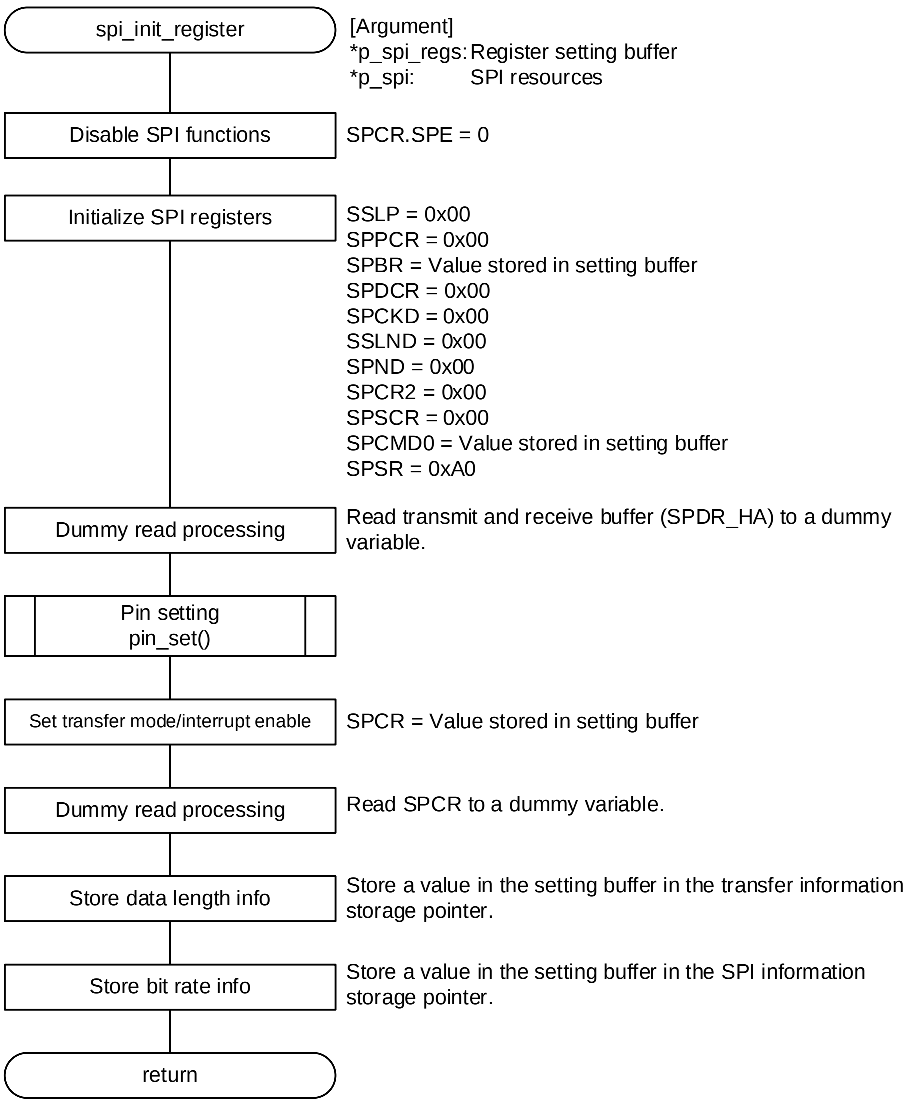

spi_init_register
[Argument]
*p_spi_regs: Register setting buffer
*p_spi: SPI resources
Disable SPI functions
SPCR.SPE = 0
Initialize SPI registers
SSLP = 0x00
SPPCR = 0x00
SPBR = Value stored in setting buffer
SPDCR = 0x00
SPCKD = 0x00
SSLND = 0x00
SPND = 0x00
SPCR2 = 0x00
SPSCR = 0x00
SPCMD0 = Value stored in setting buffer
SPSR = 0xA0
Dummy read processing
Read transmit and receive buffer (SPDR_HA) to a dummy variable.
Pin setting
pin_set()
Set transfer mode/interrupt enable
SPCR = Value stored in setting buffer
Dummy read processing
Read SPCR to a dummy variable.
Store data length info
Store a value in the setting buffer in the transfer information storage pointer.
Store bit rate info
Store a value in the setting buffer in the SPI information storage pointer.
return

Figure 4-23 spi_init_register Function Processing Flow

4.1.18 spi_interrupt_disable Function

Table 4-19 spi_interrupt_disable Function Specifications

Format	static void spi_interrupt_disable(st_spi_resources_t * const p_spi)
Description	Disables interrupts.
Argument	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources to be controlled.
Return value	None
Remarks	–

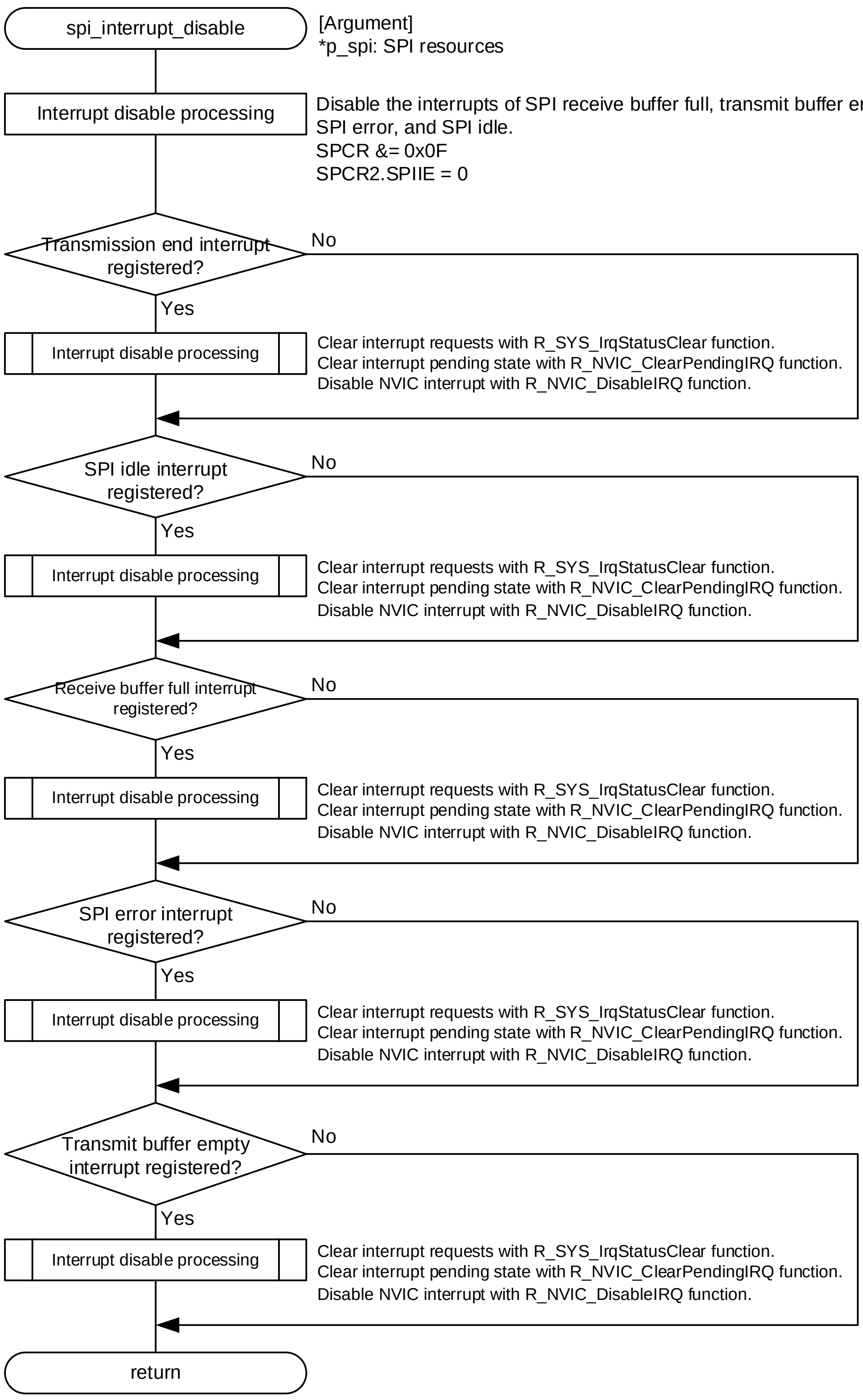

spi_interrupt_disable
[Argument]
*p_spi: SPI resources
Interrupt disable processing
Disable the interrupts of SPI receive buffer full, transmit buffer empty, SPI error, and SPI idle.
SPCR &= 0x0F
SPCR2.SPIIE = 0
Transmission end interrupt registered?
No
Yes
Interrupt disable processing
Clear interrupt requests with R_SYS_IrqStatusClear function.
Clear interrupt pending state with R_NVIC_ClearPendingIRQ function.
Disable NVIC interrupt with R_NVIC_DisableIRQ function.
SPI idle interrupt registered?
No
Yes
Interrupt disable processing
Clear interrupt requests with R_SYS_IrqStatusClear function.
Clear interrupt pending state with R_NVIC_ClearPendingIRQ function.
Disable NVIC interrupt with R_NVIC_DisableIRQ function.
Receive buffer full interrupt registered?
No
Yes
Interrupt disable processing
Clear interrupt requests with R_SYS_IrqStatusClear function.
Clear interrupt pending state with R_NVIC_ClearPendingIRQ function.
Disable NVIC interrupt with R_NVIC_DisableIRQ function.
SPI error interrupt registered?
No
Yes
Interrupt disable processing
Clear interrupt requests with R_SYS_IrqStatusClear function.
Clear interrupt pending state with R_NVIC_ClearPendingIRQ function.
Disable NVIC interrupt with R_NVIC_DisableIRQ function.
Transmit buffer empty interrupt registered?
No
Yes
Interrupt disable processing
Clear interrupt requests with R_SYS_IrqStatusClear function.
Clear interrupt pending state with R_NVIC_ClearPendingIRQ function.
Disable NVIC interrupt with R_NVIC_DisableIRQ function.
return

Figure 4-24 spi_interrupt_disable Function Processing Flow

4.1.19 spi_ir_flag_clear Function

Table 4-20 spi_ir_flag_clear Function Specifications

Format	static int32_t spi_ir_flag_clear(st_spi_resources_t * const p_spi)	
Description	Clear IR of transmit / receive interrupt to 0.	
Argument	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources to be controlled.	
Return value	ARM_DRIVER_OK	IR flag clear success
	ARM_DRIVER_ERROR_TIMEOUT	Timeout error
	A frame format error occurs if one of the following conditions is detected. • When a timeout occurs during 0 clear wait processing of the SPCR.SPRIE bit • If a timeout occurs during the 0 clear wait processing of the SPCR.SPTIE bit	
Remarks	–	

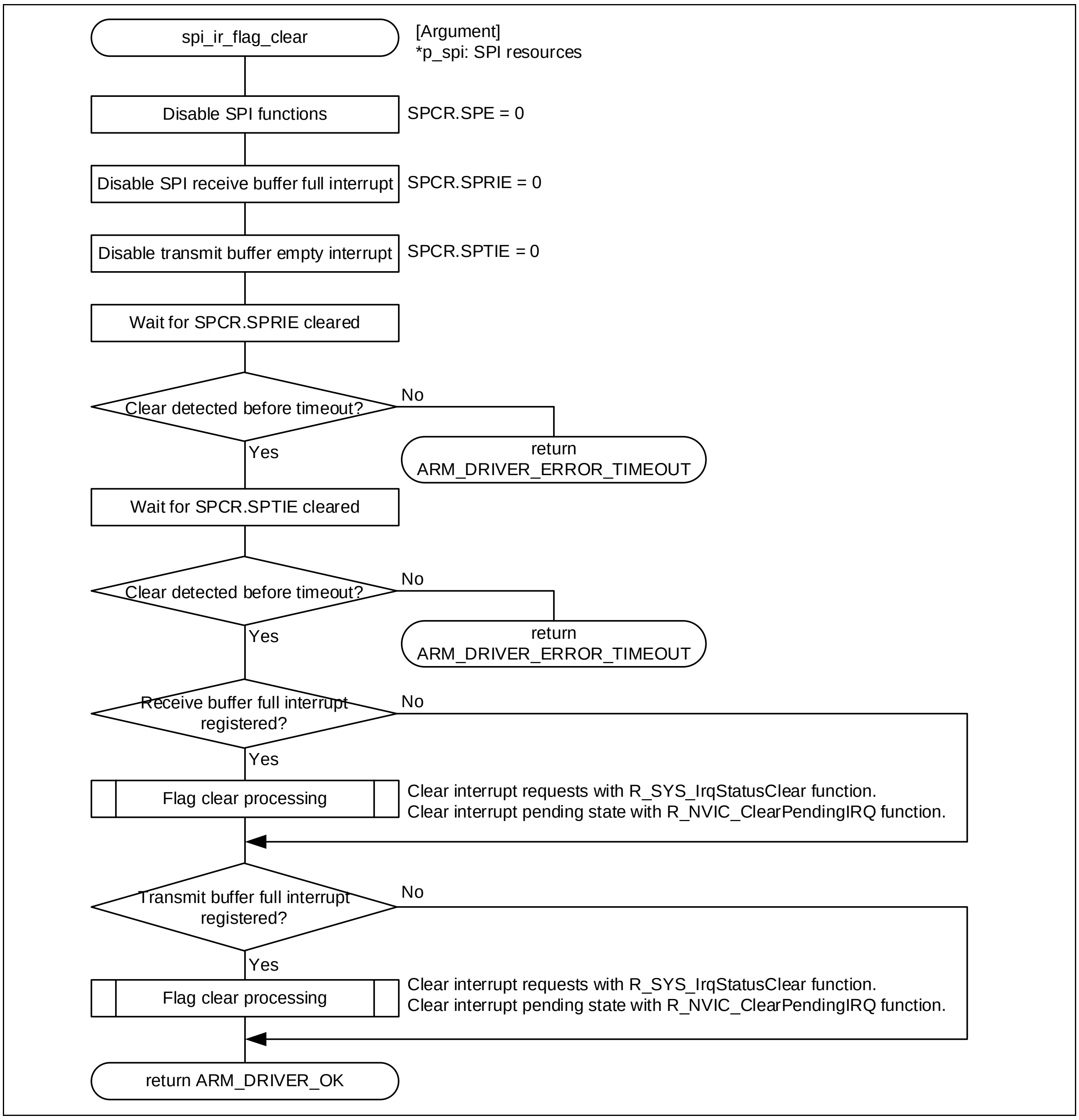
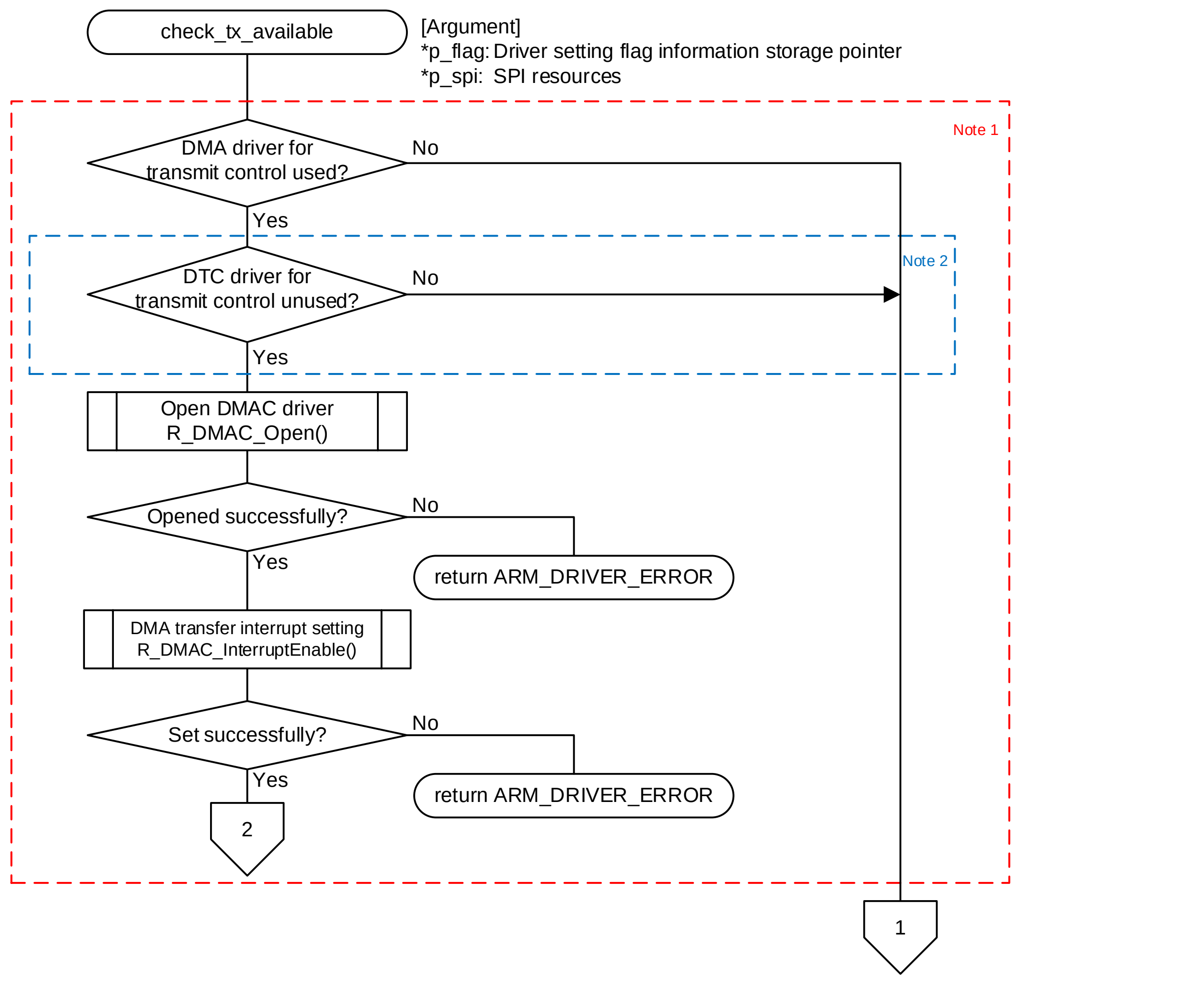

spi_ir_flag_clear
[Argument]
*p_spi: SPI resources
Disable SPI functions
SPCR.SPE = 0
Disable SPI receive buffer full interrupt
SPCR.SPRIE = 0
Disable transmit buffer empty interrupt
SPCR.SPTIE = 0
Wait for SPCR.SPRIE cleared
Clear detected before timeout?
No
return ARM_DRIVER_ERROR_TIMEOUT
Yes
Wait for SPCR.SPTIE cleared
Clear detected before timeout?
No
return ARM_DRIVER_ERROR_TIMEOUT
Yes
Receive buffer full interrupt registered?
No
Yes
Flag clear processing
Clear interrupt requests with R_SYS_IrqStatusClear function.
Clear interrupt pending state with R_NVIC_ClearPendingIRQ function.
Transmit buffer full interrupt registered?
No
Yes
Flag clear processing
Clear interrupt requests with R_SYS_IrqStatusClear function.
Clear interrupt pending state with R_NVIC_ClearPendingIRQ function.
return ARM_DRIVER_OK

Figure 4-25 spi_ir_flag_clear Function Processing Flow

4.1.20 check_tx_available Function

Table 4-21 check_tx_available Function Specifications

Format	static int32_t check_tx_available(int16_t * const p_flag, st_spi_resources_t * const p_spi)	
Description	Judges whether or not transmission is available.	
Argument	int16_t * const p_flag: Pointer to Initialization flag storage	
	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources to be controlled.	
Return value	ARM_DRIVER_OK	Transmission availability judgment completed
	ARM_DRIVER_ERROR	Transmission availability judgment failed
	Transmission availability judgment failure occurs if one of the following conditions is detected. • If an interrupt or the DTC is used in transmission and the event link setting for the SPTI interrupt fails • If an interrupt or the DTC is used in transmission and interrupt priority level setting fails • If the DTC is used in transmission and the SPTI interrupt is defined not to be used (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED) in r_system_cfg.h • If the DTC or the DMAC is used in transmission and DMA driver initialization fails • If the DMAC is used in transmission and DMAC interrupt enable setting fails • If, in a state in which the SPII interrupt is registered with NVIC in r_system_cfg.h, the event link setting for the SPII interrupt fails • If, in a state in which the SPII interrupt is registered with NVIC in r_system_cfg.h, SPII interrupt priority level setting fails • If, in a state in which the SPTEND or SPEI interrupt is registered with NVIC in r_system_cfg.h, the event link setting for the SPTEND or SPEI interrupt fails • If, in a state in which the SPTEND or SPEI interrupt is registered with NVIC in r_system_cfg.h, SPTEND or SPEI interrupt priority level setting fails	
Remarks		


check_tx_available
[Argument]
*p_flag: Driver setting flag information storage pointer
*p_spi: SPI resources
Note 1
DMA driver for transmit control used?
No
Yes
Note 2
DTC driver for transmit control unused?
No
Yes
Open DMAC driver R_DMAC_Open()
Opened successfully?
No
Yes
return ARM_DRIVER_ERROR
DMA transfer interrupt setting R_DMAC_InterruptEnable()
Set successfully?
No
Yes
return ARM_DRIVER_ERROR
2
1

Note 1. When DMAC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-26 check_tx_available Function Processing Flow (1/2)

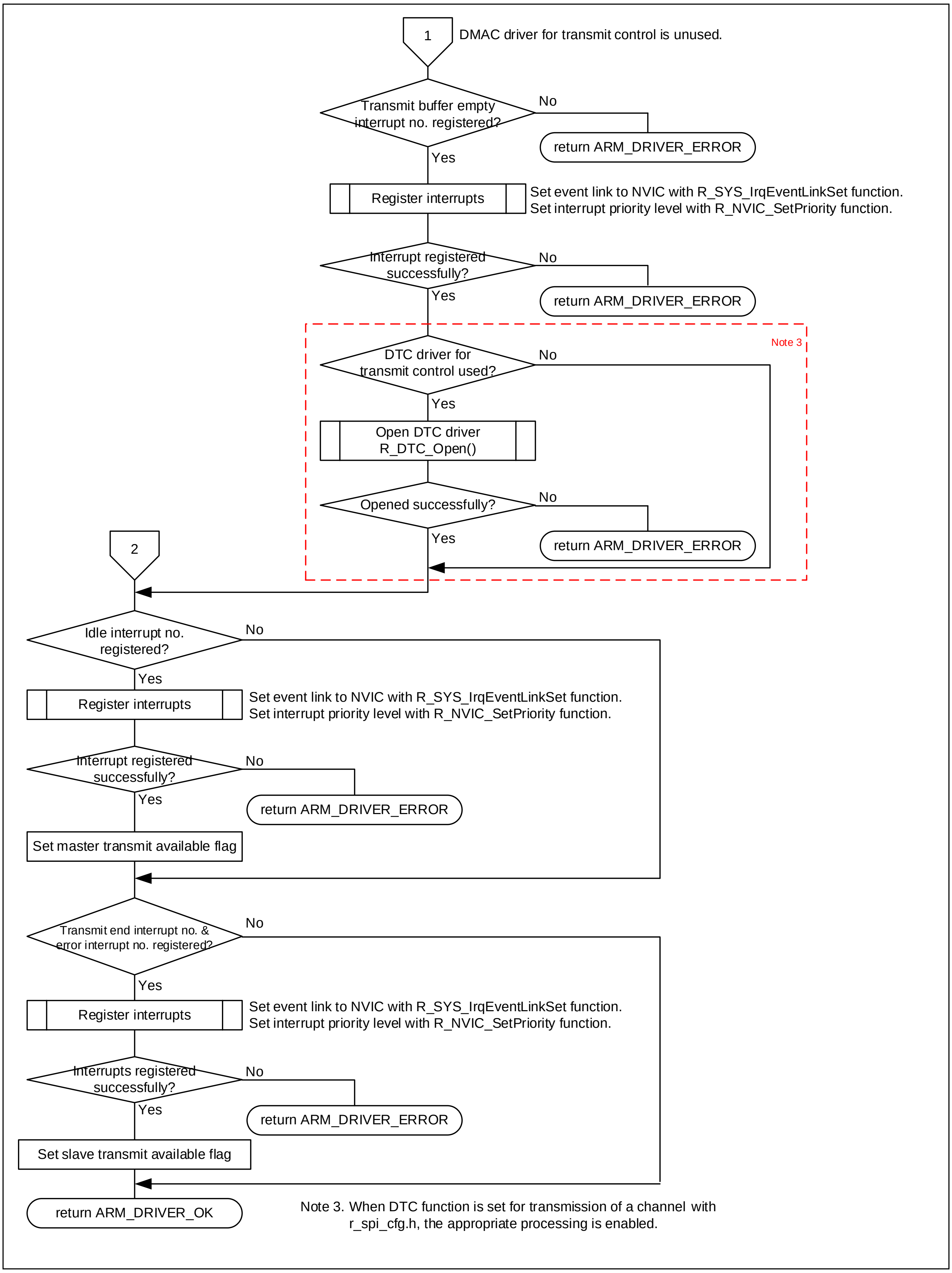
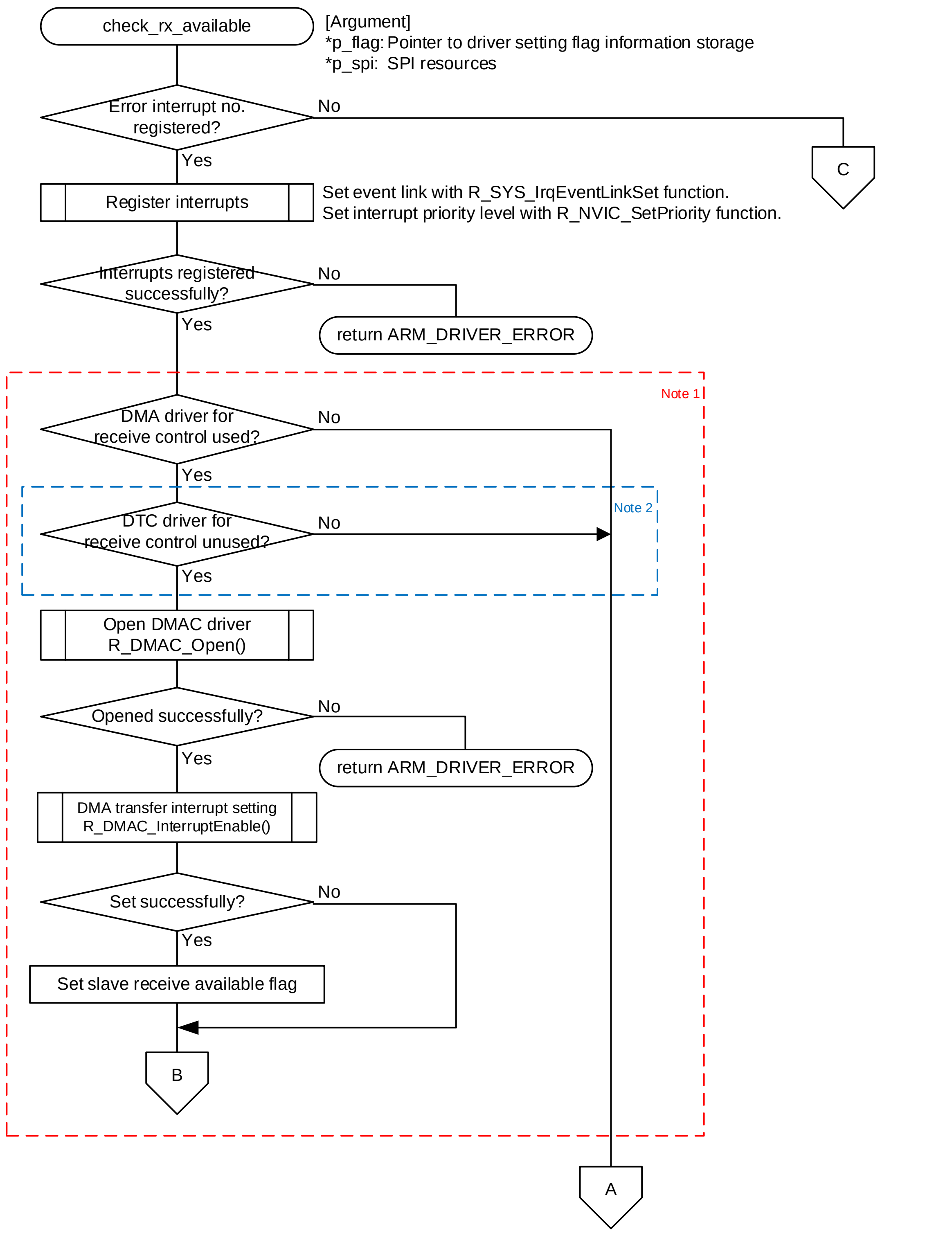

1
DMAC driver for transmit control is unused.
Transmit buffer empty interrupt no. registered?
No
return ARM_DRIVER_ERROR
Yes
Register interrupts
Set event link to NVIC with R_SYS_IrqEventLinkSet function.
Set interrupt priority level with R_NVIC_SetPriority function.
Interrupt registered successfully?
No
Yes
return ARM_DRIVER_ERROR
Note 3
DTC driver for transmit control used?
No
Yes
Open DTC driver R_DTC_Open()
Opened successfully?
No
Yes
return ARM_DRIVER_ERROR
2
Idle interrupt no. registered?
No
Yes
Register interrupts
Set event link to NVIC with R_SYS_IrqEventLinkSet function.
Set interrupt priority level with R_NVIC_SetPriority function.
Interrupt registered successfully?
No
Yes
return ARM_DRIVER_ERROR
Set master transmit available flag
Transmit end interrupt no. & error interrupt no. registered?
No
Yes
Register interrupts
Set event link to NVIC with R_SYS_IrqEventLinkSet function.
Set interrupt priority level with R_NVIC_SetPriority function.
Interrupts registered successfully?
No
Yes
return ARM_DRIVER_ERROR
Set slave transmit available flag
return ARM_DRIVER_OK
Note 3. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-27 check_tx_available Function Processing Flow (2/2)

4.1.21 check_rx_available Function

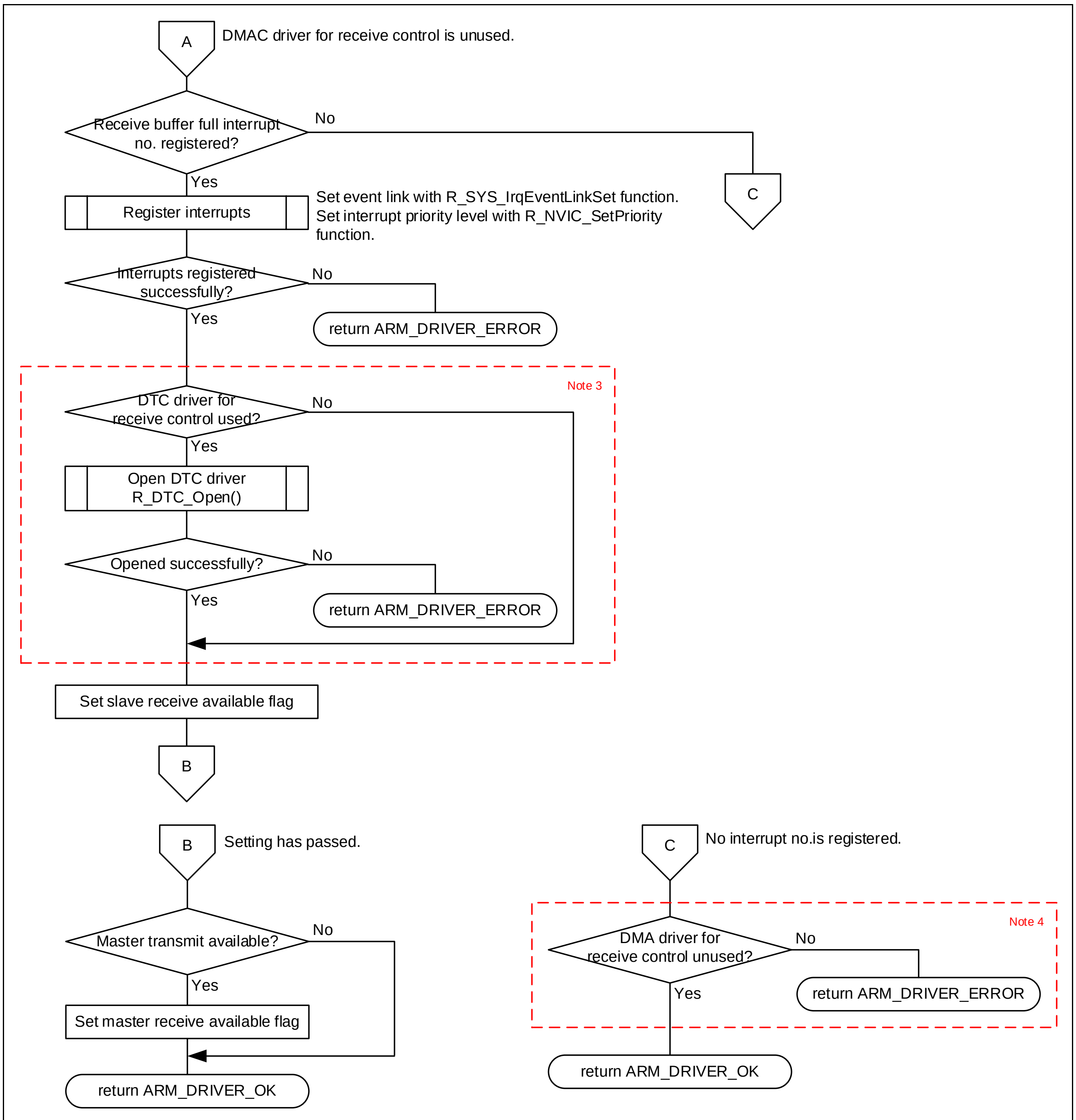
Table 4-22 check_rx_available Function Specifications

Format	static int32_t check_rx_available(int16_t * const p_flag, st_spi_resources_t * const p_spi)	
Description	Judges whether or not reception is available.	
Argument	int16_t * const p_flag: Pointer to Initialization flag storage	
	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources to be controlled.	
Return value	ARM_DRIVER_OK	Reception availability judgment completed
	ARM_DRIVER_ERROR	Reception availability judgment failed
	Reception availability judgment failure occurs if one of the following conditions is detected. • If event link setting for the SPRI or SPEI interrupt fails • If SPRI or SPEI interrupt priority level setting fails • If the DTC is used in reception and the SPRI or SPEI interrupt is defined not to be used (SYSTEM_IRQ_EVENT_NUMBER_NOT_USED) in r_system_cfg.h • If the DTC or the DMAC is used in reception and initialization of the DMA driver fails • If the DMAC is used in reception and DMAC interrupt enable setting fails	
Remarks		


check_rx_available
[Argument]
*p_flag: Pointer to driver setting flag information storage
*p_spi: SPI resources
Error interrupt no. registered?
No
C
Yes
Register interrupts
Set event link with R_SYS_IrqEventLinkSet function.
Set interrupt priority level with R_NVIC_SetPriority function.
Interrupts registered successfully?
No
Yes
return ARM_DRIVER_ERROR
Note 1
DMA driver for receive control used?
No
Yes
Note 2
DTC driver for receive control unused?
No
Yes
Open DMAC driver R_DMAC_Open()
Opened successfully?
No
Yes
return ARM_DRIVER_ERROR
DMA transfer interrupt setting R_DMAC_InterruptEnable()
Set successfully?
No
Yes
Set slave receive available flag
B
A

Note 1. When DMAC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-28 check_rx_available Function Processing Flow (1/2)


A
DMAC driver for receive control is unused.
Receive buffer full interrupt no. registered?
No
Yes
C
Register interrupts
Set event link with R_SYS_IrqEventLinkSet function.
Set interrupt priority level with R_NVIC_SetPriority function.
Interrupts registered successfully?
No
Yes
return ARM_DRIVER_ERROR
Note 3
DTC driver for receive control used?
No
Yes
Open DTC driver R_DTC_Open()
Opened successfully?
No
Yes
return ARM_DRIVER_ERROR
Set slave receive available flag
B
B
Setting has passed.
Master transmit available?
No
Yes
Set master receive available flag
return ARM_DRIVER_OK
C
No interrupt no.is registered.
Note 4
DMA driver for receive control unused?
No
Yes
return ARM_DRIVER_ERROR
return ARM_DRIVER_OK

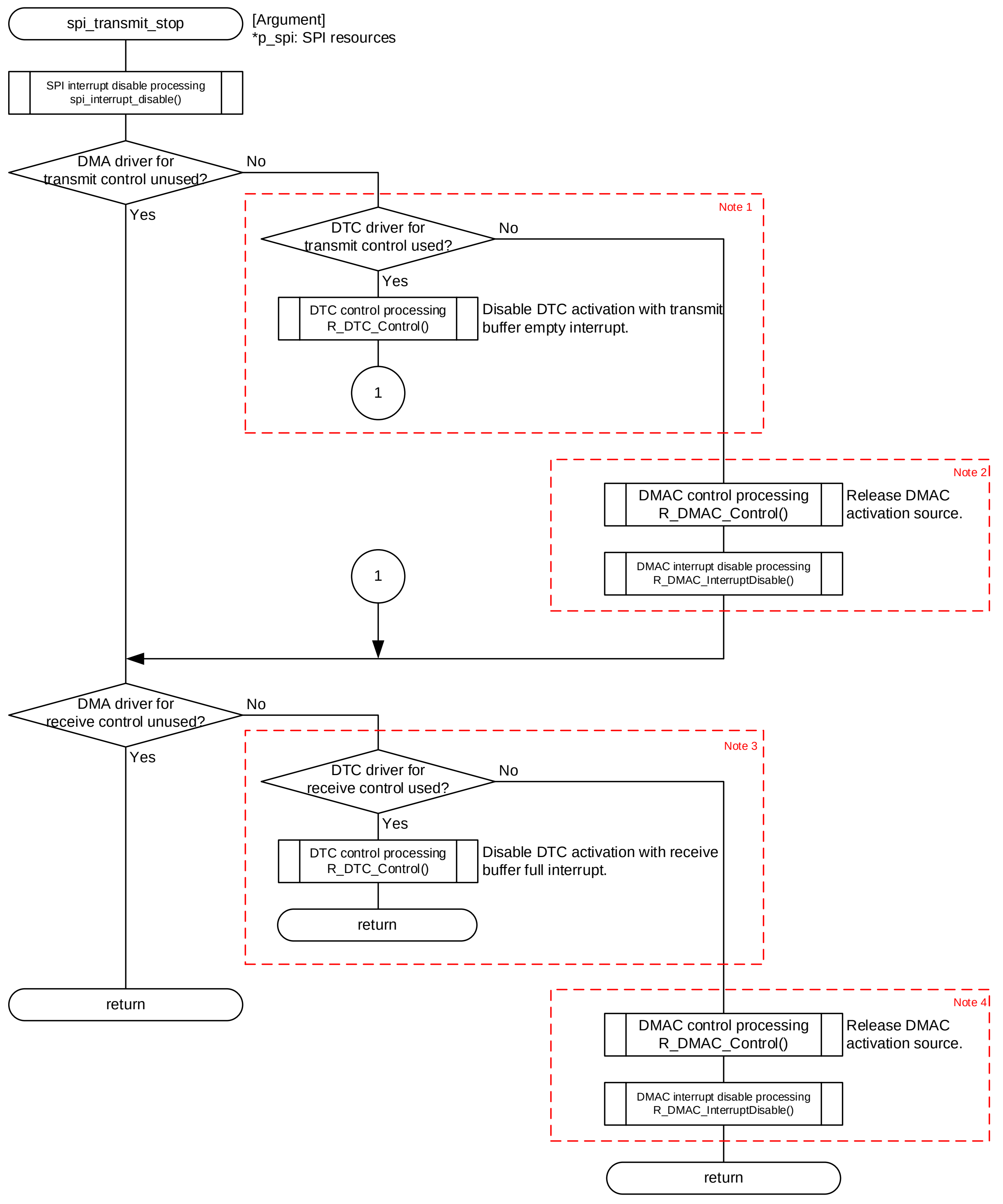
Note 3. When DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 4. When DMAC or DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-29 check_rx_available Function Processing Flow (2/2)

4.1.22 spi_transmit_stop Function

Table 4-23 spi_transmit_stop Function Specifications

Format	static void spi_transmit_stop(st_spi_resources_t const * const p_spi)
Description	Suspends transmission.
Argument	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources to be controlled.
Return value	None
Remarks	


spi_transmit_stop
[Argument]
*p_spi: SPI resources
SPI interrupt disable processing
spi_interrupt_disable()
DMA driver for transmit control unused?
No
Yes
Note 1
DTC driver for transmit control used?
No
Yes
DTC control processing
R_DTC_Control()
Disable DTC activation with transmit buffer empty interrupt.
1
Note 2
DMAC control processing
R_DMAC_Control()
Release DMAC activation source.
DMAC interrupt disable processing
R_DMAC_InterruptDisable()
1
DMA driver for receive control unused?
No
Yes
Note 3
DTC driver for receive control used?
No
Yes
DTC control processing
R_DTC_Control()
Disable DTC activation with receive buffer full interrupt.
return
return
Note 4
DMAC control processing
R_DMAC_Control()
Release DMAC activation source.
DMAC interrupt disable processing
R_DMAC_InterruptDisable()
return

Note 1. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DMAC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 3. When DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 4. When DMAC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-30 spi_transmit_stop Function Processing Flow

4.1.23 spi_send_setting Function

Table 4-24 spi_send_setting Function Specifications

Format	static int32_t spi_send_setting(void const * const p_data, uint32_t num, bool dummy_flag, st_spi_resources_t * const p_spi)	
Description	Specifies settings for transmission.	
Argument	void const * const p_data: Pointer to transmit data storage	
	uint32_t num: Transmission size	
	bool dummy_flag: Dummy transmission flag	
	st_spi_resources_t * const p_spi : SPI resources Specifies the SPI resources to be controlled.	
Return value	ARM_DRIVER_OK	Transmission setting completed
	ARM_DRIVER_ERROR	Transmission setting failed
	Transmission setting failure occurs if initialization of the DMA driver fails when the DTC or the DMAC is used for transmission.	
Remarks		

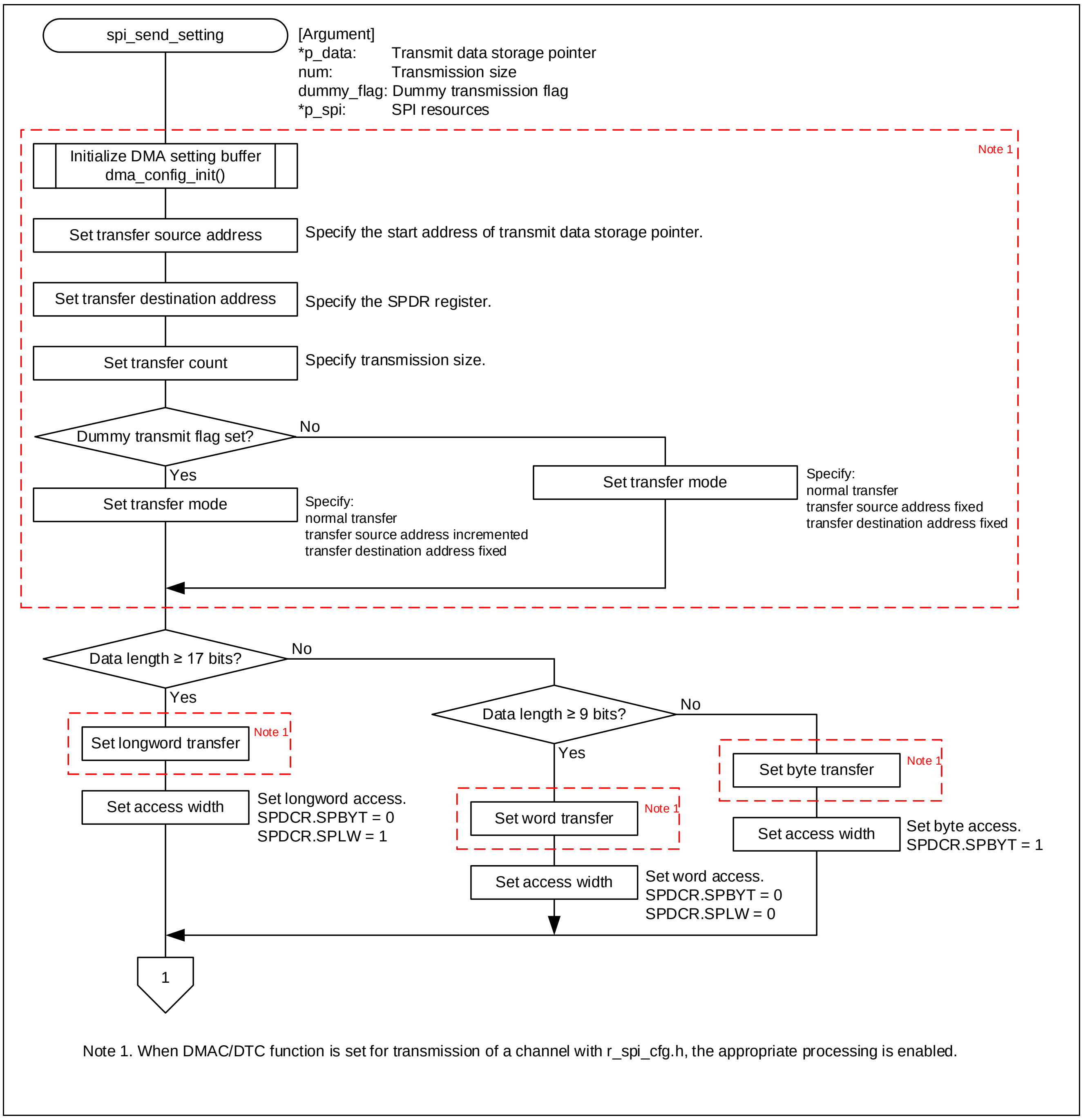
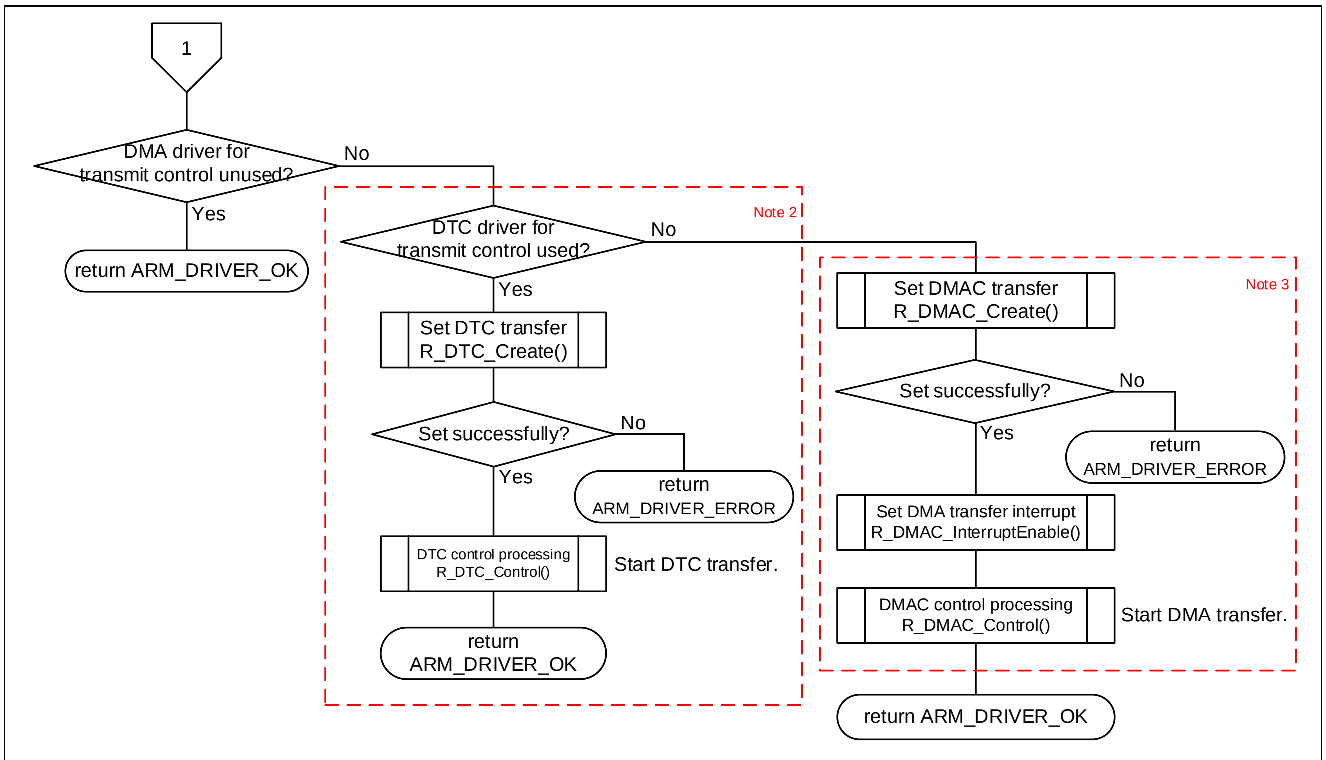

spi_send_setting
[Argument]
*p_data: Transmit data storage pointer
num: Transmission size
dummy_flag: Dummy transmission flag
*p_spi: SPI resources
Note 1
Initialize DMA setting buffer dma_config_init()
Set transfer source address
Specify the start address of transmit data storage pointer.
Set transfer destination address
Specify the SPDR register.
Set transfer count
Specify transmission size.
Dummy transmit flag set?
No
Yes
Set transfer mode
Specify: normal transfer transfer source address incremented transfer destination address fixed
Set transfer mode
Specify: normal transfer transfer source address fixed transfer destination address fixed
Data length ≥ 17 bits?
No
Yes
Set longword transfer
Note 1
Set access width
Set longword access. SPDCR.SPBYT = 0 SPDCR.SPLW = 1
Data length ≥ 9 bits?
No
Yes
Set word transfer
Note 1
Set access width
Set word access. SPDCR.SPBYT = 0 SPDCR.SPLW = 0
Set byte transfer
Note 1
Set access width
Set byte access. SPDCR.SPBYT = 1
1
Note 1. When DMAC/DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-31 spi_send_setting Function Processing Flow (1/2)


1
DMA driver for transmit control unused?
No
Yes
return ARM_DRIVER_OK
Note 2
DTC driver for transmit control used?
No
Yes
Set DTC transfer R_DTC_Create()
Set successfully?
No
Yes
return ARM_DRIVER_ERROR
DTC control processing R_DTC_Control()
Start DTC transfer.
return ARM_DRIVER_OK
Note 3
Set DMAC transfer R_DMAC_Create()
Set successfully?
No
Yes
return ARM_DRIVER_ERROR
Set DMA transfer interrupt R_DMAC_InterruptEnable()
DMAC control processing R_DMAC_Control()
Start DMA transfer.
return ARM_DRIVER_OK

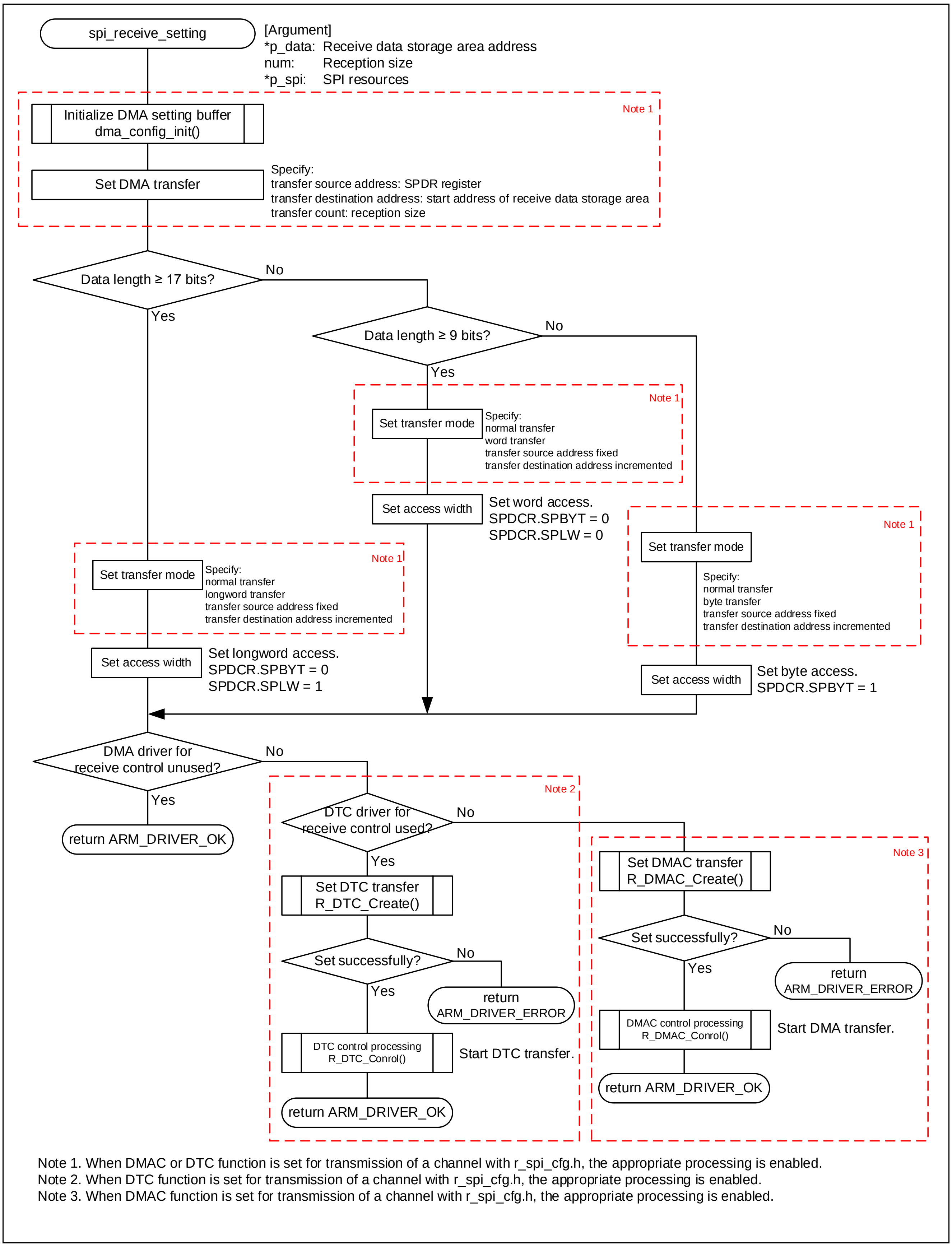
Note 2. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 3. When DMAC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-32 spi_send_setting Function Processing Flow (2/2)

4.1.24 spi_receive_setting Function

Table 4-25 spi_receive_setting Function Specifications

Format	static int32_t spi_receive_setting(void const * const p_data, uint32_t num, st_spi_resources_t * const p_spi)	
Description	Specifies settings for reception.	
Argument	void const * const p_data: Pointer to receive data storage location	
	uint32_t num: Reception size	
	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to be controlled.	
Return value	ARM_DRIVER_OK	Reception setting completed
	ARM_DRIVER_ERROR Reception setting failure occurs if initialization of the DMA driver fails when the DTC or the DMAC is used for reception.	Reception setting failed
Remarks		



Note 1. When DMAC or DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 3. When DMAC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-33 spi_receive_setting Function Processing Flow

4.1.25 dma_config_init Function

Table 4-26 dma_config_init Function Specifications

Format	static void dma_config_init(st_dma_transfer_data_cfg_t *p_cfg)
Description	Initializes the DMA driver setting structure to 0.
Argument	st_dma_transfer_data_cfg_t *p_cfg: DMA driver setting structure
Return value	None
Remarks	–

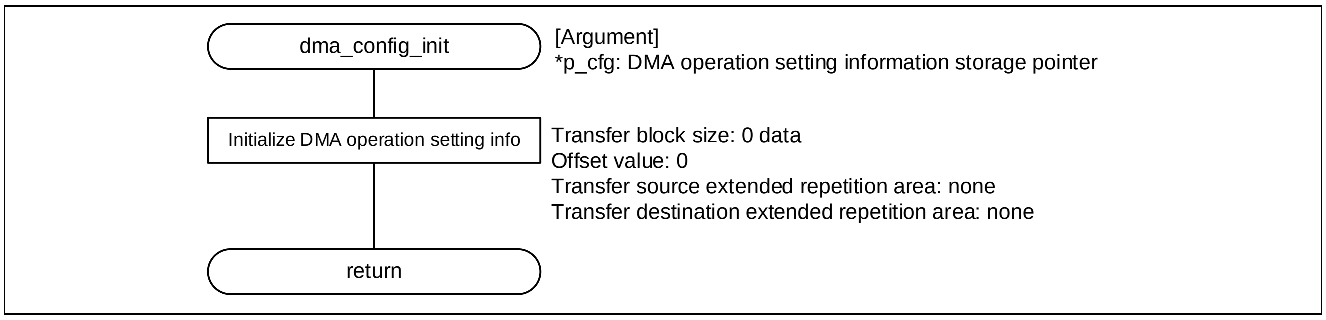

dma_config_init
[Argument]
*p_cfg: DMA operation setting information storage pointer
Initialize DMA operation setting info
Transfer block size: 0 data
Offset value: 0
Transfer source extended repetition area: none
Transfer destination extended repetition area: none
return

Figure 4-34 dma_config_init Function Processing Flow

4.1.26 spti_handler Function

Table 4-27 spti_handler Function Specifications

Format	static void spti_handler(st_spi_resources_t * const p_spi)
Description	SPTI interrupt handling processing (when using interrupts for transmission)
Argument	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to be controlled.
Return value	None
Remarks	SPTI interrupt handling processing with interrupts used for transmission

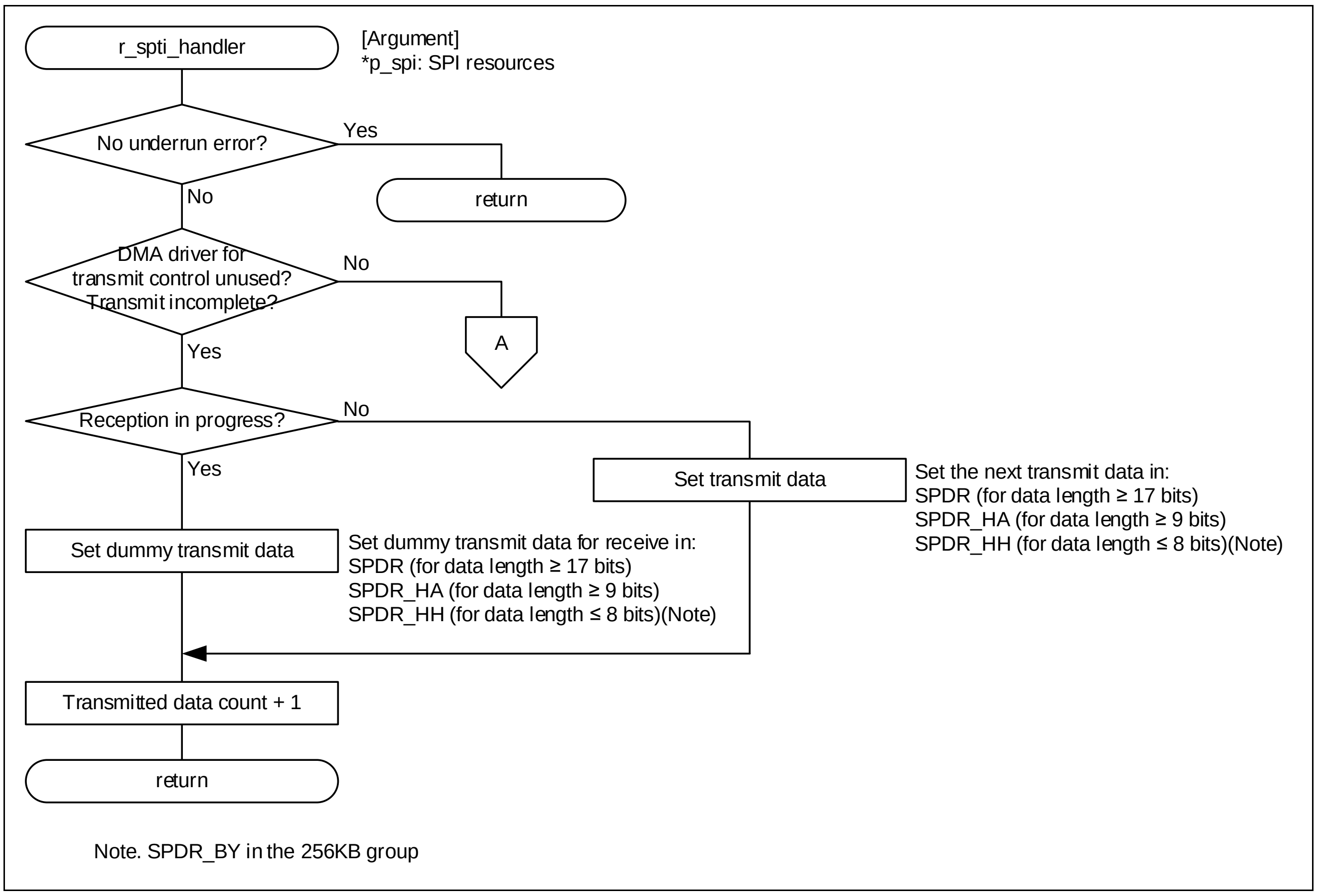
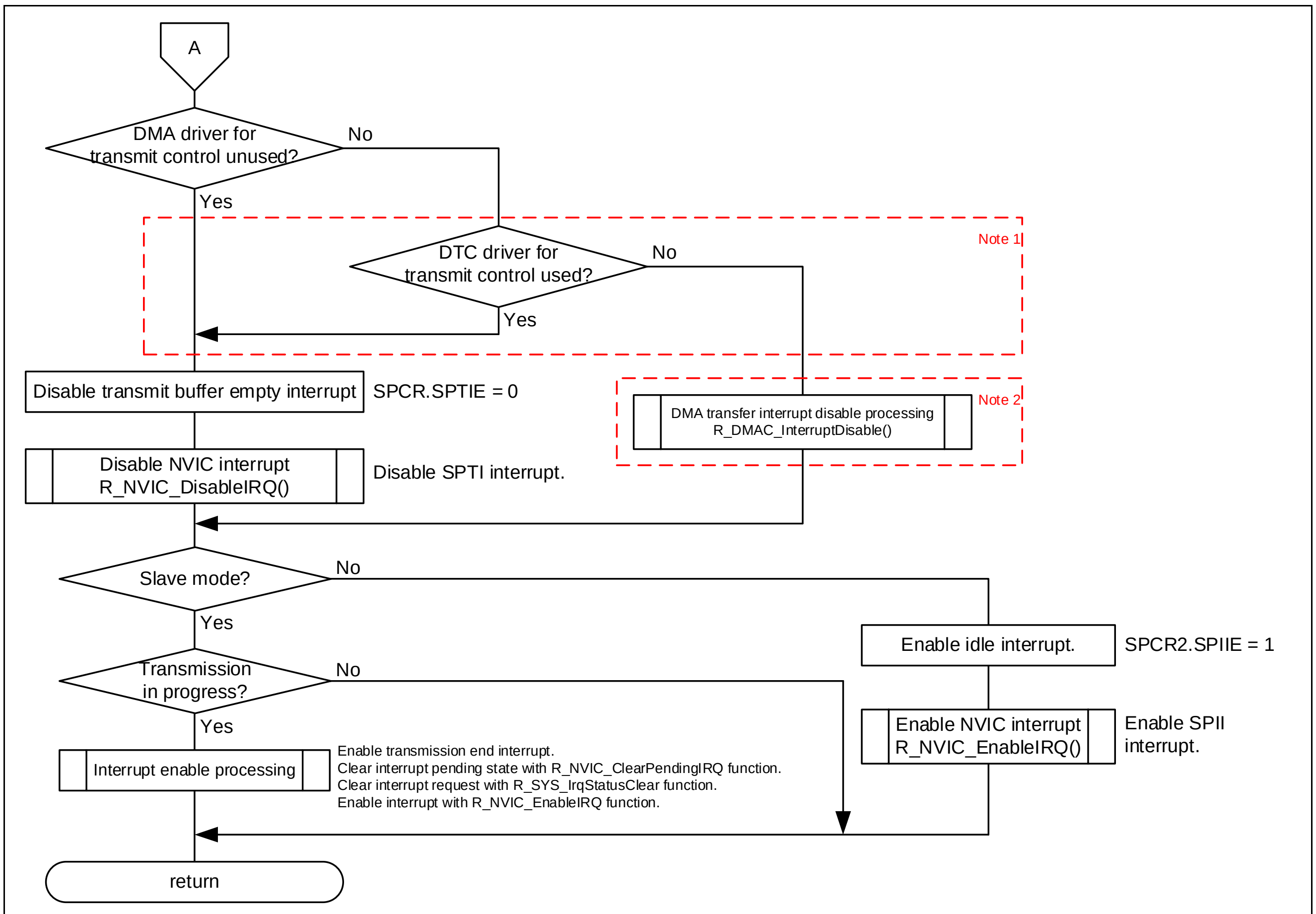

r_spti_handler
[Argument]
*p_spi: SPI resources
No underrun error?
Yes
return
No
DMA driver for transmit control unused? Transmit incomplete?
No
A
Yes
Reception in progress?
No
Yes
Set transmit data
Set the next transmit data in:
SPDR (for data length ≥ 17 bits)
SPDR_HA (for data length ≥ 9 bits)
SPDR_HH (for data length ≤ 8 bits)(Note)
Set dummy transmit data
Set dummy transmit data for receive in:
SPDR (for data length ≥ 17 bits)
SPDR_HA (for data length ≥ 9 bits)
SPDR_HH (for data length ≤ 8 bits)(Note)
Transmitted data count + 1
return
Note. SPDR_BY in the 256KB group

Figure 4-35 spti_handler Function Processing Flow (1/2)


A
DMA driver for transmit control unused?
No
Yes
Note 1
DTC driver for transmit control used?
No
Yes
Disable transmit buffer empty interrupt
SPCR.SPTIE = 0
Note 2
DMA transfer interrupt disable processing R_DMAC_InterruptDisable()
Disable NVIC interrupt R_NVIC_DisableIRQ()
Disable SPTI interrupt.
Slave mode?
No
Yes
Enable idle interrupt.
SPCR2.SPIIE = 1
Transmission in progress?
No
Yes
Enable NVIC interrupt R_NVIC_EnableIRQ()
Enable SPII interrupt.
Interrupt enable processing
Enable transmission end interrupt.
Clear interrupt pending state with R_NVIC_ClearPendingIRQ function.
Clear interrupt request with R_SYS_IrqStatusClear function.
Enable interrupt with R_NVIC_EnableIRQ function.
return

Note 1. When DTC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. When DMAC function is set for transmission of a channel with r_spi_cfg.h, the appropriate processing is enabled.

Figure 4-36 spti_handler Function Processing Flow (2/2)

4.1.27 spri_handler Function

Table 4-28 spri_handler Function Specifications

Format	static void spri_handler(st_spi_resources_t * const p_spi)
Description	SPRI interrupt handling processing (when using interrupts for reception)
Argument	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to be controlled.
Return value	None
Remarks	SPRI interrupt handling processing with interrupts used for reception

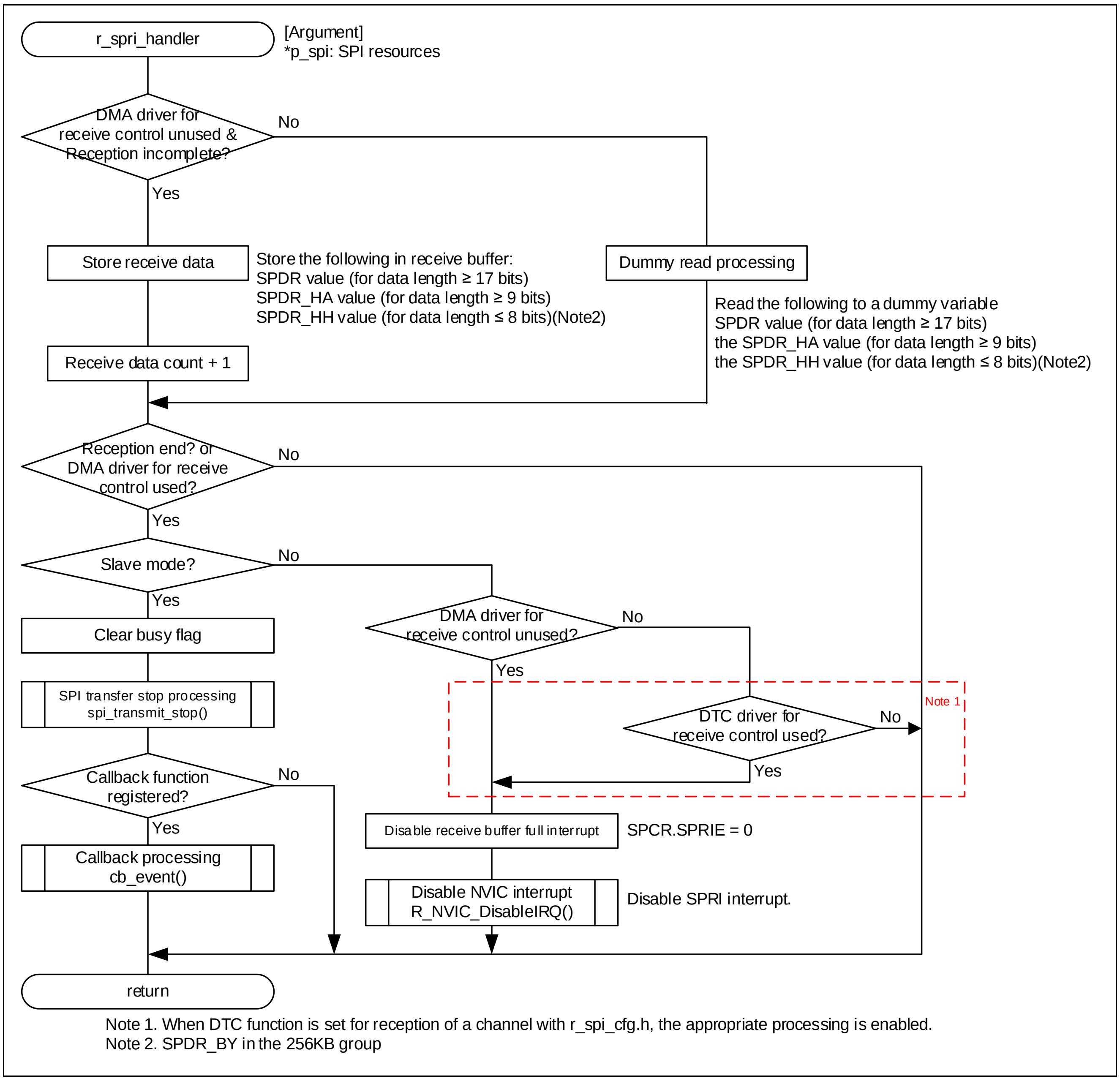

r_spri_handler
[Argument]
*p_spi: SPI resources
DMA driver for receive control unused & Reception incomplete?
No
Yes
Store receive data
Store the following in receive buffer:
SPDR value (for data length ≥ 17 bits)
SPDR_HA value (for data length ≥ 9 bits)
SPDR_HH value (for data length ≤ 8 bits)(Note2)
Dummy read processing
Read the following to a dummy variable
SPDR value (for data length ≥ 17 bits)
the SPDR_HA value (for data length ≥ 9 bits)
the SPDR_HH value (for data length ≤ 8 bits)(Note2)
Receive data count + 1
Reception end? or DMA driver for receive control used?
No
Yes
Slave mode?
No
Yes
Clear busy flag
DMA driver for receive control unused?
No
Yes
SPI transfer stop processing spi_transmit_stop()
DTC driver for receive control used?
No
Yes
Note 1
Callback function registered?
No
Yes
Disable receive buffer full interrupt
SPCR.SPRIE = 0
Callback processing cb_event()
Disable NVIC interrupt R_NVIC_DisableIRQ()
Disable SPRI interrupt.
return
Note 1. When DTC function is set for reception of a channel with r_spi_cfg.h, the appropriate processing is enabled.
Note 2. SPDR_BY in the 256KB group

Figure 4-37 spri_handler Function Processing Flow

4.1.28 spii_handler Function

Table 4-29 spii_handler Function Specifications

Format	static void spii_handler(st_spi_resources_t * const p_spi)
Description	SPEI interrupt handling processing
Argument	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to be controlled.
Return value	None
Remarks	–

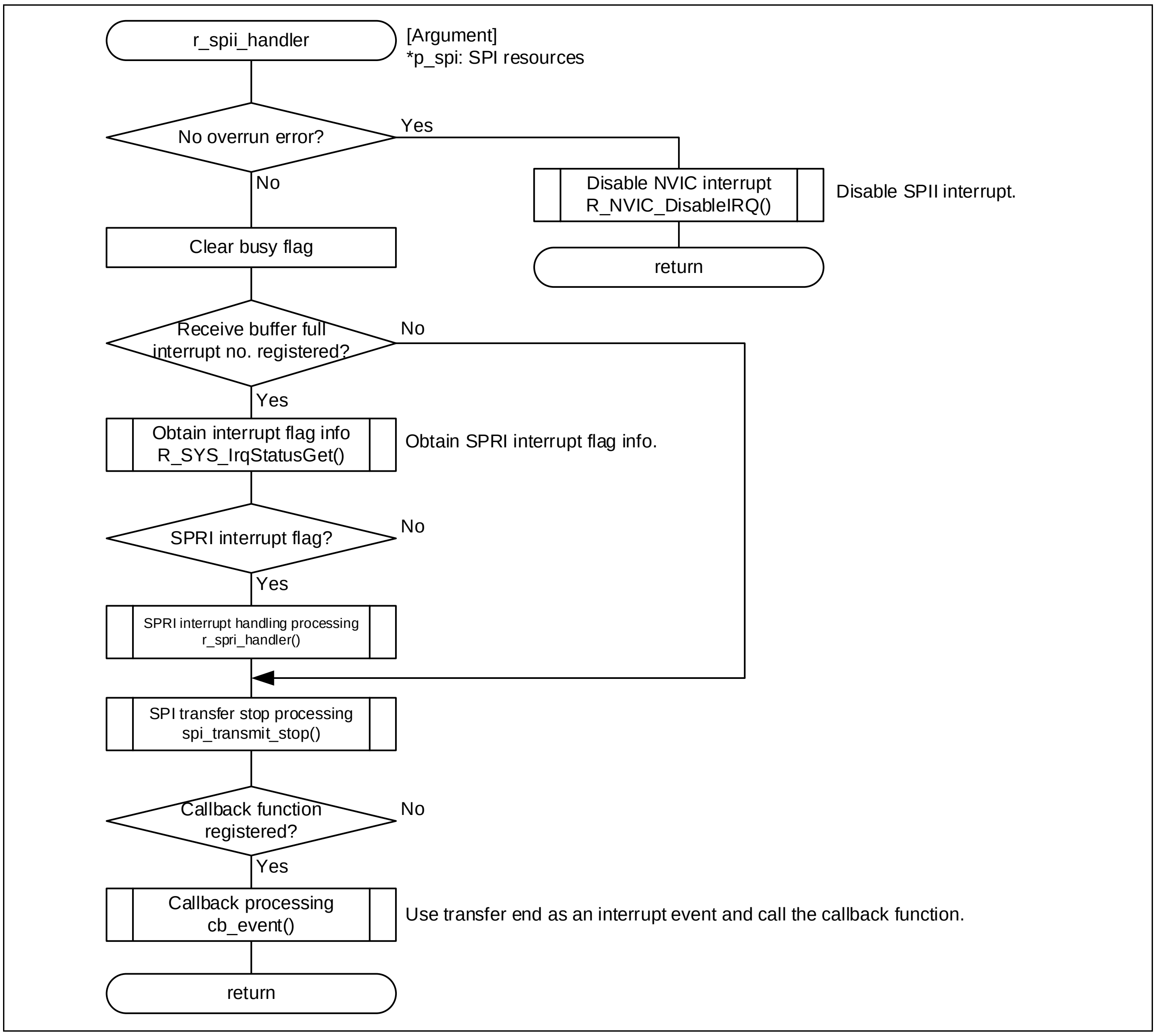


Figure 4-38 spii_handler Function Processing Flow

4.1.29 spei_handler Function

Table 4-30 pei_handler Function Specifications

Format	static void spei_handler(st_spi_resources_t * const p_spi)
Description	SPEI interrupt handling processing
Argument	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to be controlled.
Return value	None
Remarks	–

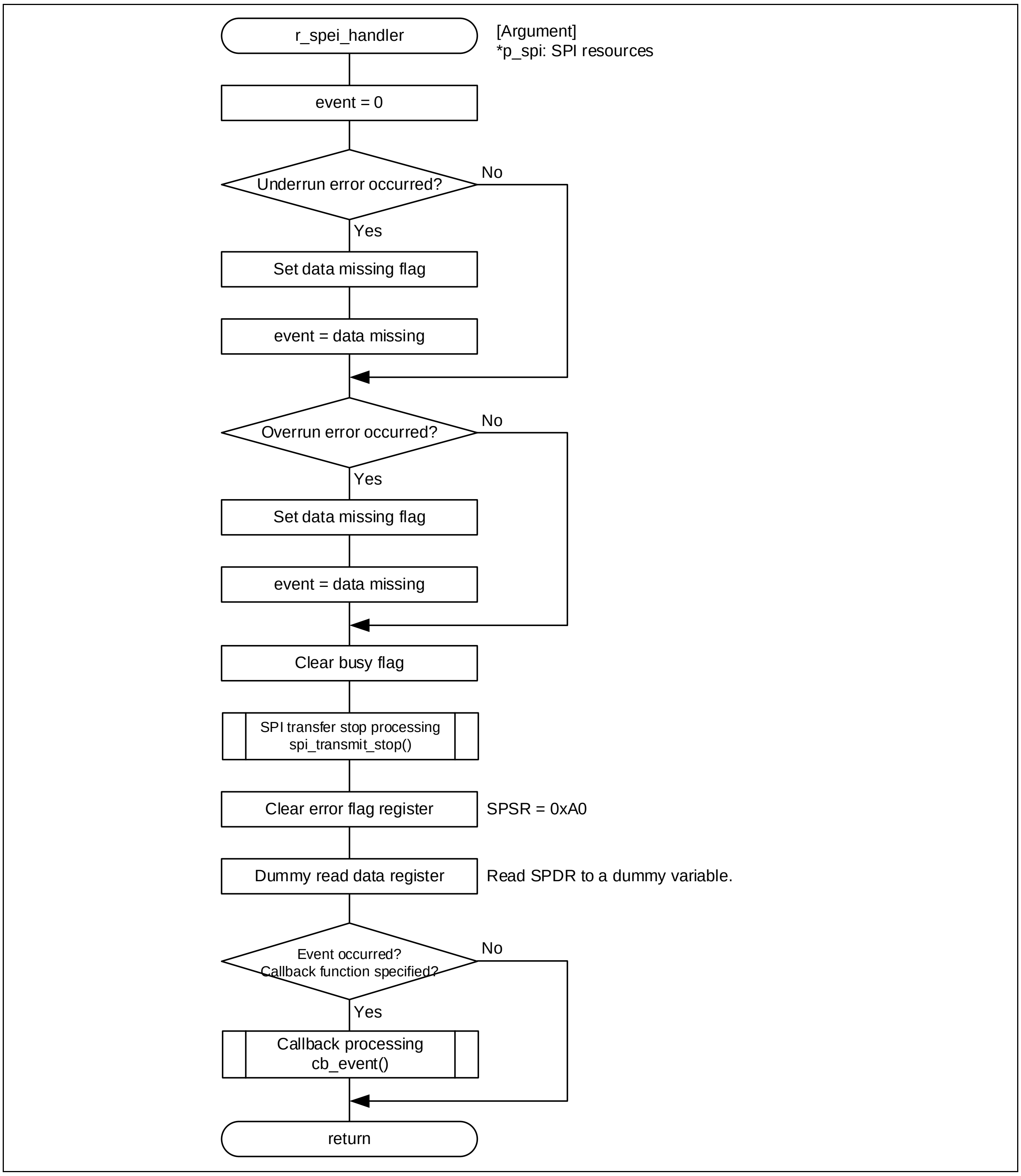


Figure 4-39 pei_handler Function Processing Flow

4.1.30 sptend_handler Function

Table 4-31 sptend_handler Function Specifications

Format	static void sptend_handler(st_spi_resources_t * const p_spi)
Description	SPEI interrupt handling processing
Argument	st_spi_resources_t * const p_spi: SPI resources Specifies the SPI resources to be controlled.
Return value	None
Remarks	–

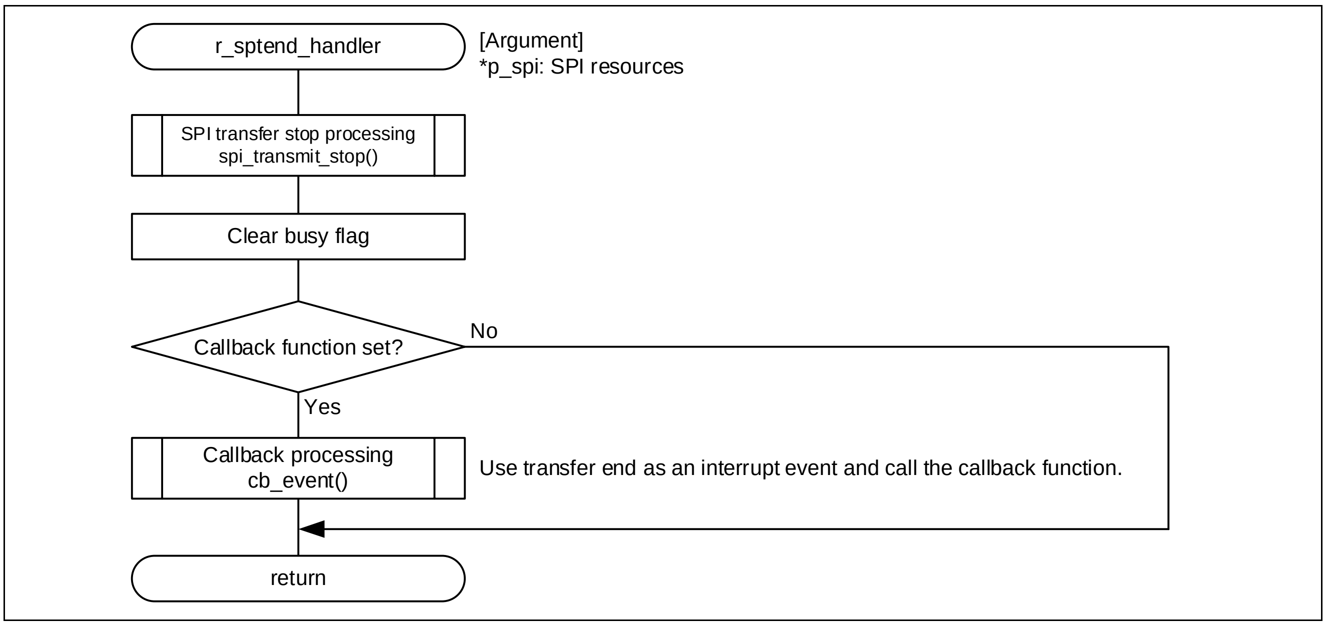


Figure 4-40 sptend_handler Function Processing Flow

4.2 Macro and Type Definitions

This section shows the definitions of the macros used in the driver and the types of them.

4.2.1 Macro Definition List

Table 4-32 List of Macro Definitions (1/2)

Macro Definition	Setting	Description
R_SPI0_ENABLE	(1)	SPI0 resource enable definition
R_SPI1_ENABLE	(1)	SPI1 resource enable definition
SPI_FLAG_INITIALIZED	(1U << 0)	SPI initialization complete flag definition
SPI_FLAG_POWERED	(1U << 1)	Module released flag definition
SPI_FLAG_CONFIGURED	(1U << 2)	Mode setting complete flag definition
SPI_FLAG_MASTER_SEND_ AVAILABLE	(1U << 3)	Master transmission enabled state flag definition
SPI_FLAG_MASTER_RECEIVE_ AVAILABLE	(1U << 4)	Master reception enabled state flag definition
SPI_FLAG_SLAVE_SEND_ AVAILABLE	(1U << 5)	Slave transmission enabled state flag definition
SPI_FLAG_SLAVE_RECEIVE_ AVAILABLE	(1U << 6)	Slave reception enabled state flag definition
SPI_SPTI0_DMAC_SOURCE_ID	(0x9A)	DELS bit settings for SPTI0
SPI_SPRI0_DMAC_SOURCE_ID	(0x99)	DELS bit settings for SPRI0
SPI_SPTI1_DMAC_SOURCE_ID	(0x9F)	DELS bit settings for SPTI1
SPI_SPRI1_DMAC_SOURCE_ID	(0x9E)	DELS bit settings for SPRI1
SPI_PRV_USED_DMAC_DTC_DRV	SPI_PRV_USED_TX_DMAC_DTC_DRV | SPI_PRV_USED_RX_DMAC_DTC_DRV	Definition for DMAC/DTC driver use judgment
SPI_PRV_USED_TX_DMAC_DTC_ DRV	SPI0_TRANSMIT_CONTROL | SPI1_TRANSMIT_CONTROL	Definition for DMAC/DTC transmission judgment
SPI_PRV_USED_RX_DMAC_DTC_ DRV	SPI0_RECEIVE_CONTROL | SPI1_RECEIVE_CONTROL	Definition for DMAC/DTC reception judgment
SPI_PRV_USED_DMAC_DRV	SPI_PRV_USED_TX_DMAC_DRV | SPI_PRV_USED_RX_DMAC_DRV	Definition for DMAC driver availability judgment
SPI_PRV_USED_TX_DMAC_DRV	SPI_PRV_USED_TX_DMAC_DTC_DRV & 0x00FF	Definition for DMAC transmission judgment
SPI_PRV_USED_RX_DMAC_DRV	SPI_PRV_USED_RX_DMAC_DTC_DRV & 0x00FF	Definition for DMAC reception judgment
SPI_PRV_USED_DTC_DRV	SPI_PRV_USED_TX_DTC_DRV | SPI_PRV_USED_RX_DTC_DRV	Definition for DTC driver availability judgment
SPI_PRV_USED_TX_DTC_DRV	SPI_PRV_USED_TX_DMAC_DTC_DRV & SPI_USED_DTC	Definition for DTC transmission judgment
SPI_PRV_USED_RX_DTC_DRV	(SPI_PRV_USED_RX_DMAC_DTC_DRV & SPI_USED_DTC	Definition for DTC reception judgment

Table 4-33 List of Macro Definitions (2/2)

Definition	Value	Description
SPI_PRV_SPCMD0_SPB_OFFSET	(8)	Offset value for setting SPCMD.SPB
SPI_PRV_SPCMD0_SPB_CLR_MASK	(0xF0FF)	Mask value for clearing SPCMD.SPB
SPI_PRV_SPCMD0_SPB_20BIT	(0x0000)	Data length setting (20 bits)
SPI_PRV_SPCMD0_SPB_24BIT	(0x0100)	Data length setting (24 bits)
SPI_PRV_SPCMD0_SPB_32BIT	(0x0200)	Data length setting (32 bits)
SPI_PRV_SPCMD0_SPB_8BIT	(0x0400)	Data length setting (8 bits)
SPI_PRV_EXEC_SEND	(0x00)	Transmit operation definition
SPI_PRV_EXEC_RECEIVE	(0x01)	Receive operation definition
SPI_PRV_EXEC_TRANSFER	(0x02)	Transmit/receive operation definition
SPI_PRV_MASK_BRDV	(0xFFF3)	Mask value for setting SPCMD0.BRDV
SPI_PRV_BASE_BIT_MASK	(0xFFFFFFFE)	Data bit length base mask for storing reception information

4.3 Structure Definitions

4.3.1 st_spi_resources_t Structure

This structure configures the resources of the SPI.

Table 4-34 st_spi_resources_t Structure

Element Name	Type	Description
*reg	volatile SPI0_Type	Shows a target SPI register.
pin_set	r_pinset_t	Function pointer for setting pins
pin_clr	r_pinclr_t	Function pointer for releasing pins
*ss_pin	volatile uint16_t	Software-controlled SSL0 pin setting (port register)
ss_pin_pos	uint8_t	Software-controlled SSL0 pin setting (pin number)
*info	st_spi_info_t	SPI status information
*xfer	st_spi_transfer_info_t	SPI transfer information
lock_id	e_system_mcu_lock_t	SPI lock ID
mstp_id	e_lpm_mstp_t	SPI module stop ID
spti_irq	IRQn_Type	SPTI interrupt number assigned in NVIC
spri_irq	IRQn_Type	SPRI interrupt number assigned in NVIC
spii_irq	IRQn_Type	SPII interrupt number assigned in NVIC
spei_irq	IRQn_Type	SPEI interrupt number assigned in NVIC
sptend_irq	IRQn_Type	SPTEND interrupt number assigned in NVIC
spti_iesr_val	uint32_t	IESR register setting for SPTI interrupt
spri_iesr_val	uint32_t	IESR register setting for SPRI interrupt
spii_iesr_val	uint32_t	IESR register setting for SPII interrupt
spei_iesr_val	uint32_t	IESR register setting for SPEI interrupt
sptend_iesr_val	uint32_t	IESR register setting for SPTEND interrupt
spti_priority	uint32_t	SPTI interrupt priority level
spri_priority	uint32_t	SPRI interrupt priority level
spii_priority	uint32_t	SPII interrupt priority level
spei_priority	uint32_t	SPEI interrupt priority level
sptend_priority	uint32_t	SPTEND interrupt priority level
*tx_dma_drv	DRIVER_DMA	DMA driver for transmission If interrupts are used for transmission, NULL is set.
tx_dma_source	uint16_t	DELS bit value set for SPTI
*tx_dtc_info	st_dma_transfer_data_t	Address at which DTC transfer information for transmission is stored
*rx_dma_drv	DRIVER_DMA	DMA driver for reception If interrupts are used for reception, NULL is set.
rx_dma_source	uint16_t	DELS bit value set for SPRI
*rx_dtc_info	st_dma_transfer_data_t	Address at which DTC transfer information for reception is stored
spti_callback	system_int_cb_t	SPTI interrupt callback function
spri_callback	system_int_cb_t	SPRI interrupt callback function
spii_callback	system_int_cb_t	SPII interrupt callback function
spei_callback	system_int_cb_t	SPEI interrupt callback function
sptend_callback	system_int_cb_t	SPTEND interrupt callback function

4.3.2 st_spi_transfer_info_t Structure

This structure is used to manage the SPI transmission/reception information.

Table 4-35 st_spi_transfer_info_t Structure

Element Name	Type	Description
num	uint32_t	Transmission/reception size
*rx_buf	void	Receive buffer
*tx_buf	void	Transmit buffer
rx_cnt	uint32_t	Reception count
tx_cnt	uint32_t	Transmission count
tx_def_val	uint16_t	Dummy transmit data
data_bits	uint8_t	Data bit length
exec_state	uint8_t	Transmission, reception, or transmission/reception status

4.3.3 st_spi_info_t Structure

This structure is used to manage the SPI information.

Table 4-36 st_spi_info_t Structure

Element Name	Type	Description
cb_event	ARM_SPI_SignalEvent_t	Callback function to be executed when an event occurs When this value is NULL, no callback function will be executed.
status	ARM_SPI_STATUS	SPI communication status
tx_status	st_spi_transfer_info_t	SPI transmission and reception information
mode	uint32_t	Operation mode ARM_SPI_MODE_INACTIVE: The SPI is not active. ARM_SPI_MODE_MASTER: Master mode operation ARM_SPI_MODE_SLAVE: Slave mode operation
bps	uint32_t	Baud rate setting
flags	uint16_t	Driver status flag b0: Driver initialization state (0: Uninitialized, 1: Initialized) b1: Module stop state (0: Module stop state, 1: Module stop released) b2: SPI mode setting complete state (0: Not set, 1: Setting completed) b3: Master transmission availability (0: Master transmission not available, 1: Master transmission available) b4: Master reception availability (0: Master reception not available, 1: Master reception available) b5: Slave transmission availability (0: Slave transmission not available, 1: Slave transmission available) b6: Slave reception availability (0: Slave reception not available, 1: Slave reception available)

4.3.4 st_spi_reg_buf_t Function

This structure is used for the register setting buffers.

Table 4-37 st_spi_reg_buf_t Structure

Element Name	Type	Description
mode	int32_t	Buffer for setting SPI operation mode
spcmd0	uint16_t	Buffer for setting SPCMD0 register
spcr	uint8_t	Buffer for setting SPCR register
spbr	uint8_t	Buffer for setting SPBR register
data_bits	uint8_t	Buffer for setting a data bit length
bps	uint32_t	Buffer for setting a baud rate

4.4 Data Table Definitions

This section shows the definitions for the main data table used for SPI driver processing.

4.4.1 Data Table for Bit Rate Division Setting

The bit rate division setting data table is a table used for setting SPCMD0.BRDV defined with type uint16_t.

Table 4-38 Bit Rate Division Setting Data Table (gs_spi_brdv_tbl)

Frequency Division Ratio	Data Table Settings	BRDV[1:0]		Description
		b3	b2	
0	0x0000	0	0	Base bit rate (Note)
2	0x0004	0	1	Base bit rate (Note) divided by 2
4	0x0008	1	0	Base bit rate (Note) divided by 4
8	0x000C	1	1	Base bit rate (Note) divided by 8

Note. The base bit rate is determined by the SPBR register value. The SPBR setting value is automatically calculated when the baud rate is set.

4.5 Calling External Functions

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

Table 4-39 External Functions Called from SPI Driver APIs and Calling Conditions (1/2)

API	Functions Called	Conditions (Note)
Initialize	R_SYS_ResourceLock	None
	R_NVIC_GetPriority	None
	R_NVIC_SetPriority	None
	R_SYS_IrqEventLinkSet	None
	R_DMAC_Open	The DMAC driver was used for transmission or reception.
	R_DMAC_InterruptEnable	
	R_DMAC_Close	The DMAC driver was used for transmission or reception and initialization failed.
	R_DTC_Open	The DTC driver was used for transmission or reception.
	R_DTC_Close	The DTC driver was used for transmission or reception and initialization failed.
Uninitialize	R_LPM_ModuleStart	The Uninitialize function was executed in the module stop state
	R_LPM_ModuleStop	None
	R_SYS_ResourceUnlock	None
	R_NVIC_ClearPendingIRQ	None
	R_SYS_IrqStatusClear	None
	R_NVIC_DisableIRQ	None
	R_RSPI_Pinclr_CHn(n = 0,1)	None
	R_RSPI_Pinset_CHn(n = 0,1)	None
	R_DMAC_Close	The DMAC driver was used for transmission or reception.
	R_DTC_Close	The DTC driver was used for transmission or reception.
PowerControl	R_LPM_ModuleStart	ARM_POWER_FULL was specified (module stop state released)
	R_RSPI_Pinclr_CHn(n = 0,1)	
	R_LPM_ModuleStop	ARM_POWER_OFF was specified (module stop state entered)
	R_NVIC_ClearPendingIRQ	
	R_SYS_IrqStatusClear	
	R_NVIC_DisableIRQ	
Send	R_NVIC_EnableIRQ	None
	R_SYS_IrqStatusClear	None
	R_NVIC_DisableIRQ	None
	R_DMAC_Create	The DMAC driver was used for transmission.
	R_DMAC_InterruptEnable	
	R_DMAC_Control	
	R_DTC_Create	The DTC driver was used for transmission.
	R_DTC_Control	

Note: If operation terminates due to a parameter check error, the functions may not be called even when no condition is specified.

Table 4-40 External Functions Called from SPI Driver APIs and Calling Conditions (2/2)

API	Functions Called	Conditions (Note)
Receive	R_NVIC_EnableIRQ	None
	R_DMAC_Create	The DMAC driver was used for reception, or the DMAC driver was used for transmission in clock synchronous mode (with transmission enabled).
	R_DMAC_InterruptEnable	
	R_DMAC_Control	
	R_NVIC_ClearPendingIRQ	None
	R_SYS_IrqStatusClear	None
	R_DTC_Create	The DTC driver was used for reception, or the DTC driver was used for transmission in clock synchronous mode (with transmission enabled).
	R_DTC_Control	
Transfer	R_NVIC_EnableIRQ	None
	R_DMAC_Create	The DMAC driver was used for transmission/reception, or the DMAC driver was used for transmission in clock synchronous mode (with transmission enabled).
	R_DMAC_InterruptEnable	
	R_DMAC_Control	
	R_NVIC_ClearPendingIRQ	None
	R_SYS_IrqStatusClear	None
	R_DTC_Create	The DTC driver was used for transmission/reception, or the DTC driver was used for transmission in clock synchronous mode (with transmission enabled).
	R_DTC_Control	
GetDataCount	R_DMAC_GetTransferByte	When a DMAC driver is used for transmission / reception processing
	R_DTC_GetTransferByte	When a DTC driver is used for transmission / reception processing
Control	R_SYS_SystemClockFreqGet	Any of the following commands was executed: • ARM_SPI_MODE_MASTER • ARM_SPI_SET_BUS_SPEED • ARM_SPI_GET_BUS_SPEED
	R_NVIC_ClearPendingIRQ	The ARM_SPI_ABORT_TRANSFER command was executed.
	R_SYS_IrqStatusClear	
	R_DMAC_Control	
	R_DMAC_InterruptDisable	
	R_NVIC_DisableIRQ	
	R_NVIC_ClearPendingIRQ	
	R_SYS_IrqStatusClear	
	R_RSPI_Pinset_CHn(n = 0,1)	Transmission or reception was enabled by either of the following commands: • ARM_SPI_MODE_MASTER • ARM_SPI_MODE_SLAVE
GetStatus	-	-
GetVersion	-	-
GetCapabilities	-	-

Note: If operation terminates due to a parameter check error, the functions may not be called even when no condition is specified.

5. Usage Notes

5.1 Registering SPI Interrupts with NVIC

It is necessary to register the interrupts used for communication control with the NVIC in r_system_cfg.h.

For details, see section 2.3, Pin Configuration.

5.2 Pin Configuration

The pins to be used by this driver are set and released respectively with the R_RSPI_Pinset_CHn (n=0,1) and R_RSPI_Pinclr_CHn functions in pin.c. The R_RSPI_Pinset_CHn function is called when transmission or reception is enabled by the Control function. The R_RSPI_Pinclr_CHn function is called when transmission and reception are disabled by the Control function, PowerControl function, or Uninitialize function.

Select the pin to be used by editing the R_RSPI_Pinset_CHn and R_RSPI_Pinclr_CHn (n = 0, 1) functions of pin.c.

Pin names of SPI function have added _A, _B, _C, and _D suffixes. When assigning SPI functions, select the functional pins with the same suffix. (Note)

Figures 5-1 to 5-3 show the coding examples for setting the pin configuration.

Note. In the case of SPI function, the same signal names are present with the suffixes “_A”, “_B” “_C” and “_D” attached. These indicate groups in terms of timing adjustment, and signals from different groups cannot be used at the same time. The exceptions are the RSPCKA_C and MOSIA_C signals for the SPI and the SSLB0_D signal, which can be used at the same time as signals from group B.

```
/*******************************************************************************//**
* @brief This function sets Pin of RSPI0.
* @note  Several pin names have added _A, _B, and _C suffixes.@n
*       When assigning the SPI functions, select the functional pins with the same suffix.@n
*       Comment out the terminal of unused suffix.@n
*       When using “RSPCKA_C, MOSIA_C” added by the SPI, select the pair of RSPCKA_B and RSPCKA_C
*       and the pair of MOSIA_B and MOSIA_C. When using “SSLB0_D” added by SPI,
*       select the pair of SSLB0_D and SSLB0_B.
**********************************************************************************/
/* Function Name : R_RSPI_Pinset_CH0 */
void R_RSPI_Pinset_CH0(void)  // @suppress("API function naming") @suppress("Function length")
{
    /* Disable protection for PFS function (Set to PWPR register) */
    R_SYS_RegisterProtectDisable(SYSTEM_REG_PROTECT_MPC);

//    /* MISOA_A : P105 */
//    PFS->P105PFS_b.PMR  = 0U;
//    PFS->P105PFS_b.ASEL = 0U;
//    PFS->P105PFS_b.ISEL = 0U;
//    PFS->P105PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P105PFS_b.PMR  = 1U;

    /* Set P500 as MISOA */
    /* MISOA_B : P500 */
    PFS->P500PFS_b.ASEL = 0U;
    PFS->P500PFS_b.ISEL = 0U;
    PFS->P500PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
    PFS->P500PFS_b.PMR  = 1U;

//    /* MOSIA_A : P104 */
//    PFS->P104PFS_b.PMR  = 0U;
//    PFS->P104PFS_b.ASEL = 0U;
//    PFS->P104PFS_b.ISEL = 0U;
//    PFS->P104PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P104PFS_b.PMR  = 1U;

    /* Set P010 as MOSIA */
    /* MOSIA_B : P010 */
    PFS->P010PFS_b.ASEL = 0U;
    PFS->P010PFS_b.ISEL = 0U;
    PFS->P010PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
    PFS->P010PFS_b.PMR  = 1U;

    /* MOSIA_C : P501 */
//    PFS->P501PFS_b.ASEL = 0U;
//    PFS->P501PFS_b.ISEL = 0U;
//    PFS->P501PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P501PFS_b.PMR  = 1U;

//    /* RSPCKA_A : P107 */
//    PFS->P107PFS_b.PMR  = 0U;
//    PFS->P107PFS_b.ASEL = 0U;
//    PFS->P107PFS_b.ISEL = 0U;
//    PFS->P107PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P107PFS_b.PMR  = 1U;

    /* Set P011 as RSPCKA */
    /* RSPCKA_B : P011 */
    PFS->P011PFS_b.ASEL = 0U;
    PFS->P011PFS_b.ISEL = 0U;
    PFS->P011PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
    PFS->P011PFS_b.PMR  = 1U;

    /* RSPCKA_C : P502 */
//    PFS->P502PFS_b.ASEL = 0U;
//    PFS->P502PFS_b.ISEL = 0U;
//    PFS->P502PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P502PFS_b.PMR  = 1U;
```

Figure 5-1   Coding Example for Setting Pin Configuration (1/3)

```
//    /* SSLA0_A : P103 */
//    PFS->P103PFS_b.ASEL = 0U;
//    PFS->P103PFS_b.ISEL = 0U;
//    PFS->P103PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P103PFS_b.PMR  = 1U;

    /* Set P012 as SSLA0 */
    /* SSLA0_B : P012 */
    PFS->P012PFS_b.ASEL = 0U;
    PFS->P012PFS_b.ISEL = 0U;
    PFS->P012PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
    PFS->P012PFS_b.PMR  = 1U;

//    /* SSLA1_A : P102 */
//    PFS->P102PFS_b.ASEL = 0U;
//    PFS->P102PFS_b.ISEL = 0U;
//    PFS->P102PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P102PFS_b.PMR  = 1U;//

    /* SSLA1_B : P013 */
//    PFS->P013PFS_b.ASEL = 0U;
//    PFS->P013PFS_b.ISEL = 0U;
//    PFS->P013PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P013PFS_b.PMR  = 1U;

//    /* SSLA2_A : P101 */
//    PFS->P101PFS_b.ASEL = 0U;
//    PFS->P101PFS_b.ISEL = 0U;
//    PFS->P101PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P101PFS_b.PMR  = 1U;

    /* SSLA2_B : P014 */
//    PFS->P014PFS_b.ASEL = 0U;
//    PFS->P014PFS_b.ISEL = 0U;
//    PFS->P014PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P014PFS_b.PMR  = 1U;

//    /* SSLA3_A : P100 */
//    PFS->P100PFS_b.ASEL = 0U;
//    PFS->P100PFS_b.ISEL = 0U;
//    PFS->P100PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P100PFS_b.PMR  = 1U;

    /* SSLA3_B : P015 */
//    PFS->P015PFS_b.ASEL = 0U;
//    PFS->P015PFS_b.ISEL = 0U;
//    PFS->P015PFS_b.PSEL = R_PIN_PRV_RSPI_PSEL;
//    PFS->P015PFS_b.PMR  = 1U;

    /* Enable protection for PFS function (Set to PWPR register) */
    R_SYS_RegisterProtectEnable(SYSTEM_REG_PROTECT_MPC);
}/* End of function R_RSPI_Pinset_CH0() */
```

Figure 5-2 Coding Example for Setting Pin Configuration (2/3)

```
/******************************************************************************//**
* @brief This function clears the pin setting of RSPI0.
*******************************************************************************/
/* Function Name : R_RSPI_Pinclr_CH0 */
void R_RSPI_Pinclr_CH0(void)  // @suppress("API function naming")
{
    /* Disable protection for PFS function (Set to PWPR register) */
    R_SYS_RegisterProtectDisable(SYSTEM_REG_PROTECT_MPC);

//    /* MISOA_A : P105 */
//    PFS->P105PFS &= R_PIN_PRV_CLR_MASK;

    /* Release MISOA pin */
    /* MISOA_B : P500 */
    PFS->P500PFS &= R_PIN_PRV_CLR_MASK;

//    /* MOSIA_A : P104 */
//    PFS->P104PFS &= R_PIN_PRV_CLR_MASK;

    /* Release MOSIA pin */
    /* MOSIA_B : P010 */
    PFS->P010PFS &= R_PIN_PRV_CLR_MASK;

    /* MOSIA_C : P501 */
//    PFS->P501PFS &= R_PIN_PRV_CLR_MASK;

//    /* RSPCKA_A : P107 */
//    PFS->P107PFS &= R_PIN_PRV_CLR_MASK;

    /* Release RSPCKA pin */
    /* RSPCKA_B : P011 */
    PFS->P011PFS &= R_PIN_PRV_CLR_MASK;

    /* RSPCKA : P502 */
//    PFS->P502PFS &= R_PIN_PRV_CLR_MASK;

//    /* SSLA0_A : P103 */
//    PFS->P103PFS &= R_PIN_PRV_CLR_MASK;

    /* Release SSLA0 pin */
    /* SSLA0_B : P012 */
    PFS->P012PFS &= R_PIN_PRV_CLR_MASK;

//    /* SSLA1_A : P102 */
//    PFS->P102PFS &= R_PIN_PRV_CLR_MASK;

    /* SSLA1_B : P013 */
//    PFS->P013PFS &= R_PIN_PRV_CLR_MASK;

//    /* SSLA2_A : P101 */
//    PFS->P101PFS &= R_PIN_PRV_CLR_MASK;

    /* SSLA2_B : P014 */
//    PFS->P014PFS &= R_PIN_PRV_CLR_MASK;

//    /* SSLA3_A : P100 */
//    PFS->P100PFS &= R_PIN_PRV_CLR_MASK;

    /* SSLA3_B : P015 */
//    PFS->P015PFS &= 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_RSPI_Pinclr_CH0() */
```

Figure 5-3 Coding Example for Setting Pin Configuration (3/3)

5.3 SSL Pin Control Using Control Function

When using the Control function (ARM_SPI_CONTROL_SS command) to control the SSL pin by software, set the pins to be used as the SSL pin using SPIn_RTS_PORT and SPIn_RTS_PIN (n=0, 1) in the r_spi_cfg.h file. For SS operation selection when selecting the operating mode using the Control function, specify ARM_SPI_SS_MASTER_SW (slave select control under software control is used during master operation) or ARM_SPI_SS_SLAVE_SW (slave select control under software control is monitored during slave operation); do not set hardware slave select control.

Figure 5-4 shows an example for setting the software controlled SSL pin to PORT107 with SPI0.

```
...
/* When using the ARM_SPI_CONTROL_SS command, cancel the following comment and set the terminal
to use */
#define SPI0_SS_PORT (PORT1->PODR)                /* Uncommented. Selects port 1 */
#define SPI0_SS_PIN (7)                              /*Selects P107 as the SS pin. */
...
```

Figure 5-4 Pin Setting Example for Using SSL Pin under Software Control

5.4 Timeout for clearing interrupt enable bit 0

The SPCR.SPRIE bit and SPCR.SPTIE bit 0 clear wait processing timeout time in the spi_ir_flag_clear function is defined by SYSTEM_CFG_API_TIMEOUT_COUNT in r_system_cfg.h. To change the timeout time, change the value of SYSTEM_CFG_API_TIMEOUT_COUNT in r_system_cfg.h (Note). An example of setting the timeout time is shown in Figure 5-5.

Note. SYSTEM_CFG_API_TIMEOUT_COUNT is a common definition in the RE01 group CMSIS software package. The waiting time for changing register settings other than this driver is also changed.

```
/*******************************************************************************************//**
 * @brief Time-out value of API until register value is changed.@n
 *******************************************************************************************/
#define SYSTEM_CFG_API_TIMEOUT_COUNT              (0x10000000)
```

Figure 5-5 SYSTEM_CFG_API_TIMEOUT_COUNT setting example

5.5 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”.

5.6 Resuming communication in slave mode and CPHA0

When CPHA (clock phase) is set to 0 (data sampling at the rising edge, data change at the falling edge) in slave mode, resuming communication during the last half of RSPCK cycle may cause incorrect operations such as underrun error or bit shifts.

After calling the callback function or after ensuring that SPI status is ready by using GetStatus function, wait for half RSPCK cycle before restarting the communication.

Figure 5-6 shows the example of restarting communication in slave mode when CPHA0 is set.

```
static uint8_t tx_data[3] = {0x01, 0x02, 0x03};

/****************************************************************************************************
* callback function
****************************************************************************************************/
static void spi_callback(uint32_t event)
{
    switch( event )
    {
        case ARM_SPI_EVENT_TRANSFER_COMPLETE:
        {
            /* RSPCK half cycle wait (5us for communication speed 100kbps) */
            R_SYS_SoftwareDelay(5, SYSTEM_DELAY_UNITS_MICROSECONDS);
            /* restart */
            spi0Drv->Send(&tx_data[0], 3);
        }
        break;

        case ARM_SPI_EVENT_DATA_LOST:
        default:
        {
            /* Describes processing when a communication error occurs */
        }
        break;
    }

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

Figure 5-6 Example of restarting communication in slave mode and CPHA0

5.7 SSL signal control during multiple data communication

In this driver, the SSL signal is negated ("H") for each data communication. If you want to keep the SSL signal at the active level ("L") until all data communication is completed during multiple data communication, use clock synchronous communication (3-wire type) and control the SSL signal with software. Figure 5-7 shows an example of SSL signal software control operation during multiple data communication.

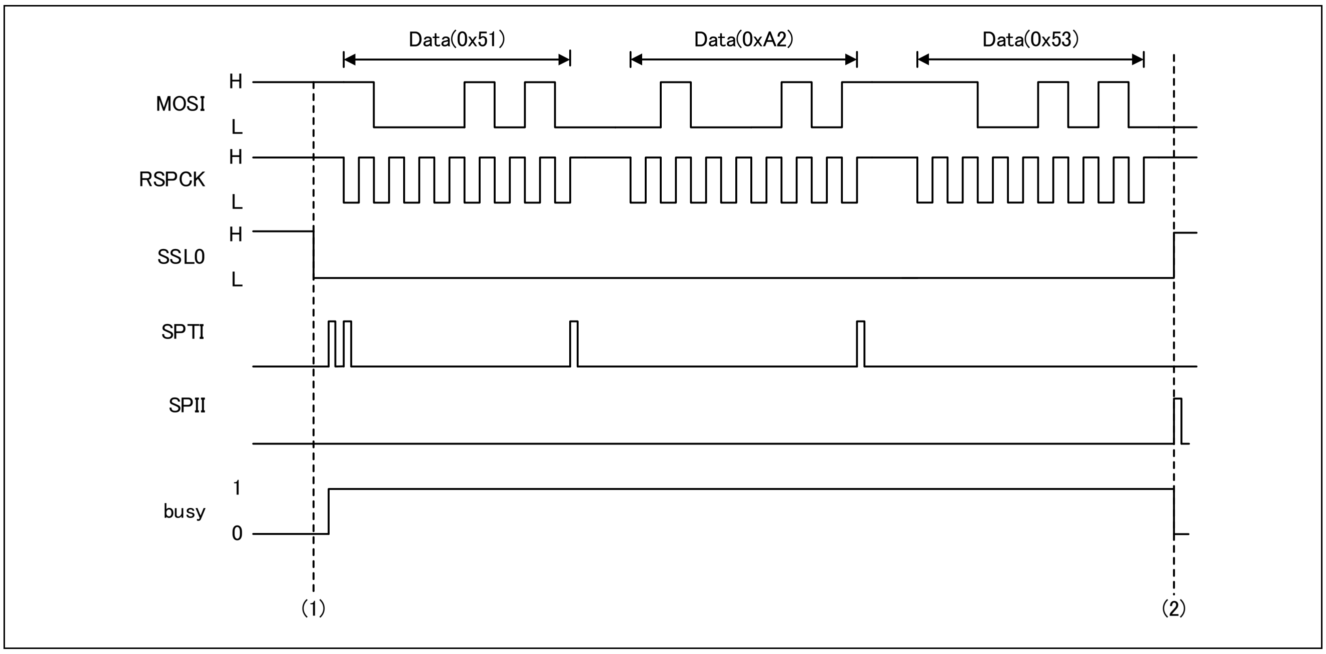


Figure 5-7 Example of SSL signal software control operation during multiple data communication

(1) When the SSL signal is set to active with the ARM_SPI_CONTROL_SS command (ARM_SPI_SS_ACTIVE) of the Control function, the SSL signal becomes "L".

(2) When all data communication is completed, if the SSL signal is set to inactive using the ARM_SPI_CONTROL_SS command (ARM_SPI_SS_INACTIVE) of the Control function, the SSL signal becomes "H".

5.8 Notes on using DTC

When DTC is selected for send control or receive control in the r_spi_cfg.h file of this driver, the R_DTC_Close function of the DTC driver is executed if the conditions are satisfied with the APIs shown in Table 5-1. When the R_DTC_Close function is executed, all DTC transfer factors are released.

When DTC is used by multiple drivers, all DTC settings are cancelled when the R_DTC_Close function is executed, causing a problem with the DTC transfer to stop.

Table 5-1 shows the conditions for executing the R_DTC_Close function in the API processing of this driver.

Table 5-1 List of functions to execute R_DTC_Close function

Function	R_DTC_Close function execution conditions
ARM_SPI_Initialize	In case of ARM_DRIVER_ERROR
ARM_SPI_Uninitialize	When executing the ARM_SPI_Uninitialize function
ARM_SPI_Send	In case of ARM_DRIVER_ERROR
ARM_SPI_Receive	In case of ARM_DRIVER_ERROR
ARM_SPI_Transfer	In case of ARM_DRIVER_ERROR

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	Dec.02.2019	12, 119~123	Modification to comment out default pin setting of pin.c
1.03	Feb.27.2020	7~10, 16,80 Program	Modified description of SPI control command (bit order definition) Fixed a bug that the setting was reversed for the bit order specified by the SPI control command (bit order definition)
1.04	Mar.5.2020	— Program (256KB)	Compatible with 256KB group The 256KB group specifications are shown below. ・ 8-bit access SPI data register name change (Change from SPDR_HH to SPDR_BY)
1.05	Apr.17.2020	Program	Changed the configuration so that it can be built without DMAC and DTC drivers.
1.06	Jun.10.2020	—	Error correction
1.07	Nov.05.2020	125 —	Added "5.8 Notes on using DTC" 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/.