

RZ/A2M Group

Simplified SD Memory Card Driver: Installation Guide

Introduction

This document provides information on installing and using the Simplified SD memory card driver.

The RZ/A2M Simplified SD memory card driver (hereinafter called “SD driver”) is a software library for Renesas Electronics original 32-bit RISC microcomputer RZ/A2M Group. SD driver enables the file operations to the SD memory card and MMC card by combining with Generic FAT filesystem software library.

It also supports the SDIO card. Reading the Card Common Control Registers (CCCR) and the Function Basic Registers (FBR), and I/O control by the I/O Read/Write Commands are enabled after the SDIO card is mounted.

Target Device

RZ/A2M

Symbols and Terminology

In this manual, it explains by using the symbols and terms shown in below as long as there is no explanation specifically.

Table 1 Symbol

Item	Descriptions
Numeric	Indicates a decimal number unless otherwise indicated in this manual.
0x	Indicates a hexadecimal number unless otherwise indicated in this manual.
0b	Indicates a binary number unless otherwise indicated in this manual.

Table 2 Terminology

Item	Descriptions
FAT	File allocation table
exFAT	Extended FAT
SD memory card	Secure digital memory card
MMC card	Multi Media Card
SDIO card	SDIO (SD Input/Output) Card
SDIO Combo card	SDIO function and SD memory function are combined.
Card	SD memory card, MMC card, and SDIO card
Default-Speed card	SD memory card supporting an SD clock maximum frequency of 25 MHz as specified by Physical Layer Specification, ver. 1.10
High-Capacity card	SD memory card with the memory capacity that exceeds 2 GB (32 GB max.) specified by Physical Layer Specification, ver. 2.00
Standard-Capacity card	The SD memory card with the memory capacity of 2 GB or less. Especially, the SD memory cards conformed to Physical Layer Specification, ver. 1.01 and ver. 1.10 are all belonged to the Standard-Capacity card.
eXtended-Capacity card	SD memory card with memory capacity that exceeds 32 GB (2 TB max.), specified by Physical Layer Specification, ver. 3.00

Contents

1.	SD Driver Outline.....	7
1.1	Function Outline	7
1.2	Program Development Procedure.....	8
2.	Software Configuration	9
3.	Application Development Procedure.....	10
3.1	Library Function List	10
3.2	Application Development Procedure	12
3.2.1	Outline	12
3.2.2	Making Device Driver Function	12
3.2.3	Making Target CPU Interface Function	13
3.3	Memory for Library Use	14
3.3.1	Library Function Work Area and Buffer Area	14
3.3.2	Specification of Library Function Work and Buffer Areas	14
3.4	Status Confirmation Method.....	15
3.4.1	Status Confirmation Method.....	15
3.4.2	Card Swapping Detection.....	15
3.4.3	SD Protocol Control Status Confirmation by Interrupt.....	16
3.4.4	SD Protocol Control Status Confirmation by Polling	17
3.5	SDCLK Decision Method.....	18
3.5.1	Clock Frequency Dividing Ratio	18
3.5.2	Stop of Clock	18
3.6	Sector Data Transfer Method	18
3.6.1	Transferring Method with Software	18
3.6.2	Transferring Method with DMA.....	18
3.7	High-Capacity Card and eXtended-Capacity Card Support.....	19
3.7.1	Selecting for High-Capacity Card Support and eXtended-Capacity Card Support.....	19
3.7.2	Obtaining Operation Mode	19
3.8	SDIO Card Support	20
3.8.1	Selecting SDIO Card Support	20
3.8.2	Obtaining Operation Mode	20
3.8.3	Detecting SDIO Interrupts	20
3.8.4	Detecting SDIO Interrupts by SD Host Controller Interrupt.....	21
3.8.5	Detecting SDIO Interrupts by Software Polling	22
3.9	SDIO Block Data Transfer Method.....	23
3.9.1	Software Transfer Method.....	23
3.9.2	DMA Transfer Method	23
3.10	Speed-up of Data Transfer Processing.....	24

3.10.1	Card Format	24
3.10.2	Data Transfer Size (SD Memory Card)	24
3.10.3	Data Transfer Size (SDIO Card)	25
3.11	Card Mount Processing	26
3.12	Error Codes	28
4.	Function Reference	30
4.1	Function Reference Details	30
4.2	Library Function	31
4.2.1	sd_init	33
4.2.2	sd_finalize	35
4.2.3	sd_set_buffer	36
4.2.4	sd_cd_int	37
4.2.5	sd_check_media	38
4.2.6	sd_set_seccnt	39
4.2.7	sd_get_seccnt	40
4.2.8	sd_mount	41
4.2.9	sd_unmount	44
4.2.10	sd_inactive	45
4.2.11	sd_read_sect	46
4.2.12	sd_write_sect	47
4.2.13	sd_get_type	49
4.2.14	sd_get_size	51
4.2.15	sd_iswp	52
4.2.16	sd_stop	53
4.2.17	sd_set_intcallback	54
4.2.18	sd_int_handler	55
4.2.19	sd_check_int	56
4.2.20	sd_get_reg	57
4.2.21	sd_get_rca	60
4.2.22	sd_get_sdstatus	61
4.2.23	sd_get_error	62
4.2.24	sd_set_cdtime	63
4.2.25	sd_set_responsetime	64
4.2.26	sd_get_ver	65
4.2.27	sd_lock_unlock	66
4.2.28	sd_get_speed	68
4.2.29	sdio_get_cia	70
4.2.30	sdio_get_ioocr	71
4.2.31	sdio_get_ioinfo	72
4.2.32	sdio_reset	73

4.2.33	sdio_set_enable	74
4.2.34	sdio_get_ready	75
4.2.35	sdio_set_blocklen	76
4.2.36	sdio_get_blocklen	77
4.2.37	sdio_set_int	78
4.2.38	sdio_get_int	79
4.2.39	sdio_read_direct	80
4.2.40	sdio_write_direct	81
4.2.41	sdio_read	82
4.2.42	sdio_write	83
4.2.43	sdio_enable_int	84
4.2.44	sdio_disable_int	84
4.2.45	sdio_set_intcallback	85
4.2.46	sdio_int_handler	85
4.2.47	sdio_check_int	86
4.2.48	sdio_abort	87
4.2.49	sdio_set_blkcnt	88
4.2.50	sdio_get_blkcnt	89
4.3	Target CPU Interface Functions	90
4.3.1	sddev_init	91
4.3.2	sddev_finalize	92
4.3.3	sddev_power_on	93
4.3.4	sddev_power_off	94
4.3.5	sddev_read_data	95
4.3.6	sddev_write_data	97
4.3.7	sddev_get_clockdiv	99
4.3.8	sddev_set_port	101
4.3.9	sddev_int_wait	102
4.3.10	sddev_loc_cpu	103
4.3.11	sddev_unl_cpu	104
4.3.12	sddev_init_dma	105
4.3.13	sddev_wait_dma_end	106
4.3.14	sddev_disable_dma	107
4.3.15	sddev_reset_dma	108
4.3.16	sddev_finalize_dma	109
4.3.17	sddev_cmd0_sdio_mount	110
4.3.18	sddev_cmd8_sdio_mount	110
4.4	Device Driver Functions	111
4.4.1	disk_status	112
4.4.2	disk_initialize	114
4.4.3	disk_read	117

4.4.4	disk_write.....	118
4.4.5	disk_ioctl.....	119
4.4.6	get_fatime.....	120
5.	Configuration Options.....	121
6.	Restrictions for Application Making.....	122
6.1	Notes for Using SD Driver.....	122
7.	Sample Program.....	123
7.1	Function Overview.....	123
7.2	Operating Environment.....	126
7.3	Confirmed Operating Conditions.....	127
7.4	Pin Names and Functions.....	129
7.5	Memory Size.....	131
7.6	Installation Procedure.....	132
7.7	Note.....	142
8.	Procedure to add the component by Smart Configurator.....	143
8.1	Component addition.....	143
8.2	Configuration Settings.....	145
8.2.1	SD card detection option settings.....	145
8.2.2	SD write protection signal detection option settings.....	146
8.2.3	SD card detection callback function settings.....	147
8.3	Pin settings.....	148
8.3.1	CD pin and WP pin settings.....	148
8.3.2	PD_1 pin (SDVcc_SEL) settings.....	149
8.3.3	PJ_1 pin (SW3 key input) settings.....	150
8.3.4	PC_1 pin (LED1 (Yellowish-green)) settings.....	151
8.4	Code generation.....	152
9.	Reference Documents.....	153
	Revision History.....	154

1. SD Driver Outline

This chapter describes the outline of Simplified SD driver software library.

1.1 Function Outline

SD driver is composed of the library function group to access the SD memory card and the SDIO card. The features of SD driver are shown as follows.

SD Driver Features

- Compact software configuration based on the combination with filesystem.
- It supports to the SD memory card from 4 MB to 2 TB.
- It supports to the MMC card from 4 MB to 2 GB.
- The software transfer or the DMA transfer can be selected as the data transfer method.
- The CPU dependence part is separated as the target CPU interface function.
- High-Capacity card support
- eXtended-Capacity card support
- Memory-saving configuration
- Bus speed mode is default speed fixed. (Supports High-Speed only for SDIO cards)
- Supports mounting the SDIO card.
- Supports reading the Card Common Control Registers (CCCR) of the SDIO card.
- Supports to read the Function Basic Register (FBR) of the SDIO card.
- Supports to read the Card Information Structure (CIS) of the SDIO card.
- Supports the SDIO card I/O control using the IO_RW_DIRECT Command (CMD52).
- Supports the SDIO card I/O control using the IO_RW_EXTENDED_Command (CMD53)
- Supports the SDIO Interrupts.

1.2 Program Development Procedure

A development flowchart of an application program that uses the SD driver is shown in Figure 1.1. To develop an application using the SD driver, a FAT filesystem library is necessary. Moreover, it is necessary to create a device driver source file and a target CPU interface source file, as shown in the figure. Refer to Chapter 3 for details of device driver functions and target CPU functions.

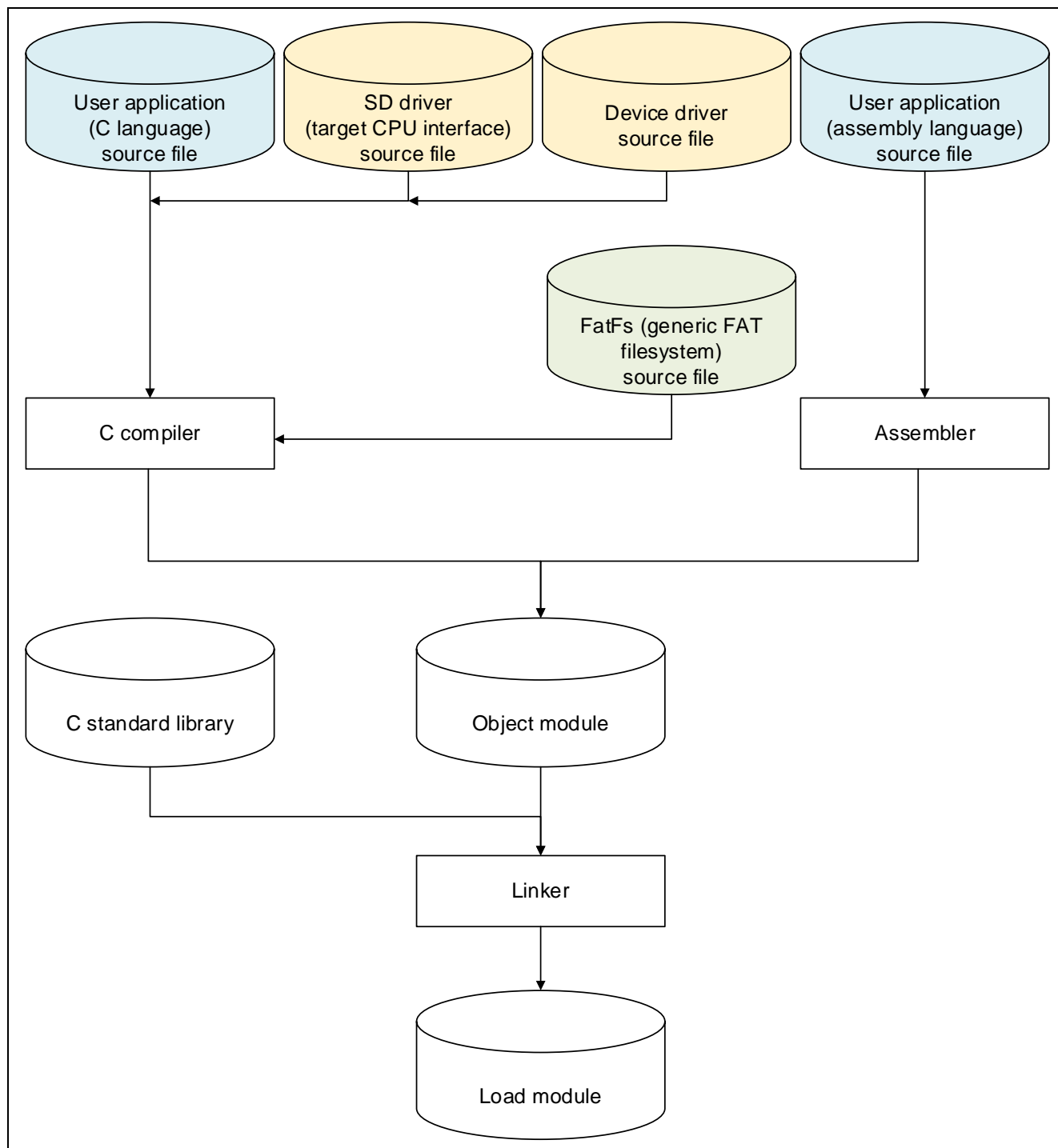


Figure 1.1 Application Program Development Flowchart

2. Software Configuration

Figure 2.1 shows the software configuration of the SD driver.

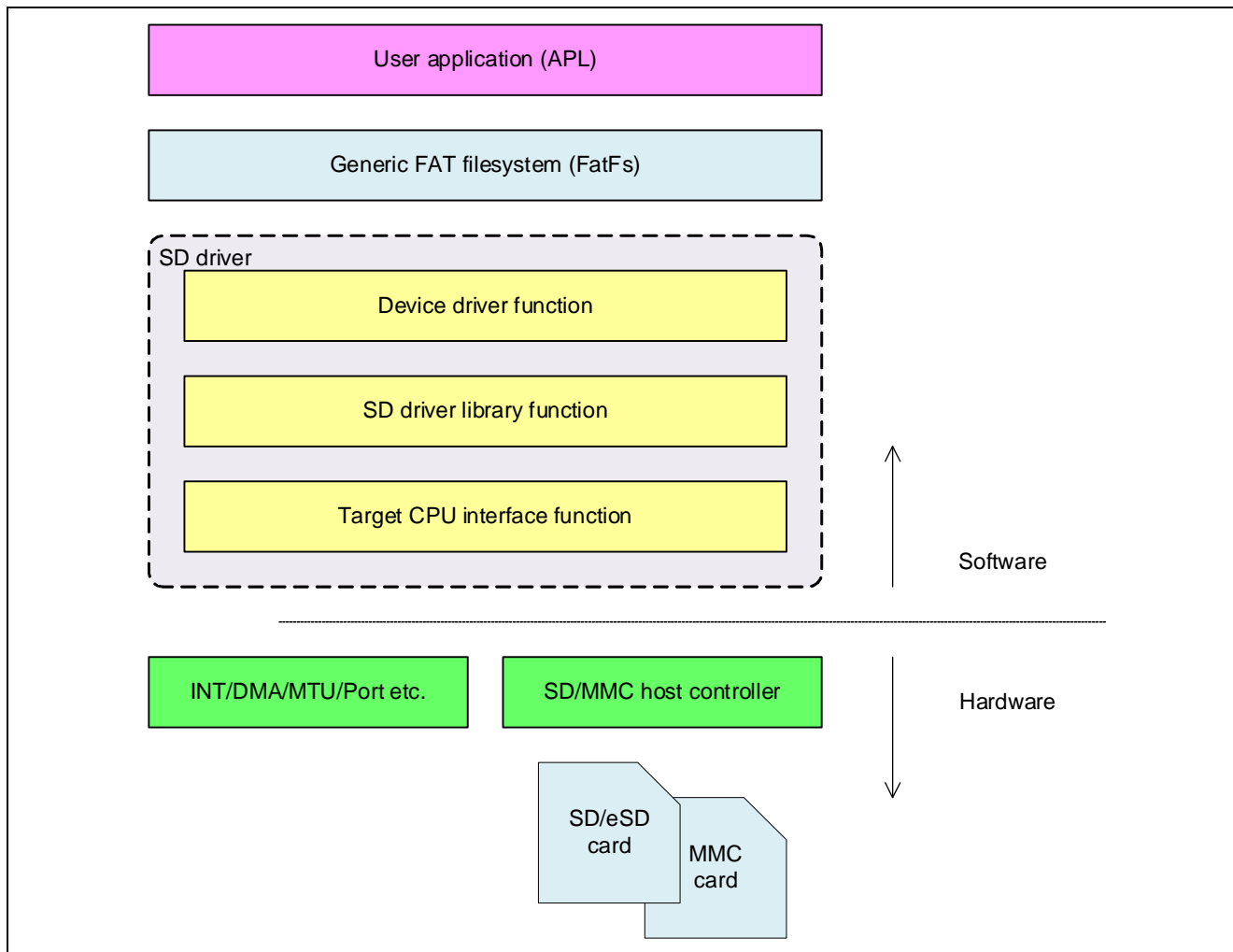


Figure 2.1 Software Configuration Diagram

3. Application Development Procedure

This chapter describes the development procedure of the application program that uses SD driver and the built-in method to the system.

3.1 Library Function List

The library functions of the SD driver are shown in Table 3.1 to Table 3.2.

Table 3.1 SD Driver Library Functions (1/2)

Function Name	Function Outline
sd_init	Initialization of SD memory card driver
sd_finalize	Termination of SD memory card driver
sd_set_buffer	Buffer area setting for library
sd_cd_int	Card swapping interrupt setting
sd_check_media	Confirmation of card insertion
sd_set_seccnt	Continuous transfer sector count setting
sd_get_seccnt	Continuous transfer sector count acquisition
sd_mount	Card mounting
sd_unmount	Card mount releasing
sd_inactive	Card disabling
sd_read_sect	Sector read from SD card
sd_write_sect	Sector write to SD card
sd_get_type	Card type and operation mode acquisition
sd_get_size	Card size acquisition
sd_iswp	Card write-protect state acquisition
sd_stop	Forced termination of card processing
sd_set_intcallback	Registration of protocol status confirmation interrupt callback function
sd_int_handler	Card interrupt handler
sd_check_int	Card interrupt request confirmation
sd_get_reg	Card register acquisition
sd_get_rca	RCA register acquisition
sd_get_sdstatus	SD status acquisition
sd_get_error	Driver error acquisition
sd_set_cdtime	Card detection time setting
sd_set_responsetime	Response timeout time setting
sd_get_ver	Library version acquisition
sd_lock_unlock	Card locking/unlocking
sd_get_speed	Card speed acquisition

Table 3.2 SD Driver Library Functions (2/2)

Function Name	Function Outline
sdio_get_cia	Obtaining SDIO card CIA information
sdio_get_ioocr	Obtaining SDIO card IO_OCR information
sdio_get_ioinfo	Obtaining SDIO card IO information
sdio_reset	SDIO reset
sdio_set_enable	SDIO function enable
sdio_get_ready	SDIO Ready confirmation
sdio_set_blocklen	SDIO block size setting
sdio_get_blocklen	Obtaining SDIO block size
sdio_set_int	SDIO interrupt setting
sdio_get_int	Obtaining SDIO interrupt setting status
sdio_read_direct	Direct-read from SDIO card
sdio_write_direct	Direct-write to SDIO card
sdio_read	Reading data from SDIO card
sdio_write	Writing data to SDIO card
sdio_enable_int	SDIO interrupt enable (SDHI module)
sdio_disable_int	SDIO interrupt disable (SDHI module)
sdio_set_intcallback	Registration of SDIO interrupt callback function
sdio_int_handler	SDIO interrupt handler
sdio_check_int	SDIO interrupt request confirmation
sdio_abort	SDIO driver termination
sdio_set_blkcnt	Continuous transfer block count setting
sdio_get_blkcnt	Continuous transfer block count acquisition

3.2 Application Development Procedure

This section describes the application development procedure when the FAT filesystem for the SD memory card is constructed by using SD driver for the target system.

3.2.1 Outline

It is necessary to make the following program that depends on the application system to construct the FAT filesystem by using SD driver.

(a) Device Driver Function

It is a function to build in SD driver as the device driver of FAT filesystem software library. Functions for performing operations such as initializing, reading from, and writing to the device are created as device driver functions. For detailed descriptions of device driver functions, refer to 4.4, Device Driver Functions.

(b) Target CPU Interface Function

This function consists of H/W initialization, power supply to SD card and so on. The interrupt controller setting and the function for the data transfer are made. For detailed descriptions of target CPU interface functions, refer to 4.3, Target CPU Interface Functions.

3.2.2 Making Device Driver Function

The device driver function is a function to build SD driver into FAT filesystem. In the device driver function, the SD driver operation mode is set and the work area that SD driver's library function uses is retained.

The device driver function list of SD driver for the case that the SD driver is built into generic FAT filesystem (FatFs)*1 is shown in Table 3.3. When it is built into the other filesystem, the device driver function should be created according to the specification of filesystem used.

For detailed descriptions of device driver functions, refer to 4.4, Device Driver Functions.

Note 1. To construct the FAT filesystem with the eXtended-Capacity card, the FAT filesystem supporting exFAT is necessary.

Table 3.3 Driver Interface Functions

Classification	Function Name	Function Outline
Device driver function	disk_status	Device status acquisition
	disk_initialize	Device initialization
	disk_read	Reading sector data (logical sector units)
	disk_write	Writing sector data (logical sector units)
	disk_ioctl	Control of other device
	get_fattime	Date and time acquisition

3.2.3 Making Target CPU Interface Function

The target CPU interface function is a function that interfaces with an internal resource of CPU that the library function of SD driver uses. It sets the port control, the interrupt controller, and the timer, etc. The target CPU interface function is called from the library function of SD driver.

The target CPU interface function list of SD driver is shown in Table 3.4.

For detailed descriptions of target CPU interface functions, refer to 4.3, Target CPU Interface Functions.

Table 3.4 Target CPU Interface Functions

Classification	Function Name	Function Outline
Target CPU interface function	sddev_init	Initialization of hardware
	sddev_finalize	Termination of hardware
	sddev_power_on	Starting of power supply to card
	sddev_power_off	Stopping of power supply to card
	sddev_read_data	Data read processing
	sddev_write_data	Data write processing
	sddev_get_clockdiv	Clock frequency dividing ratio acquisition
	sddev_set_port	Port setting for card
	sddev_int_wait	Card interrupt standby
	sddev_loc_cpu	Card interrupt disable
	sddev_unl_cpu	Card interrupt enable
	sddev_init_dma	Data transfer DMA initialization
	sddev_wait_dma_end	Data transfer DMA transfer completion standby
	sddev_disable_dma	Data transfer DMA disable
	sddev_reset_dma	Reset of DMA
	sddev_finalize_dma	Termination of DMA
	sddev_cmd0_sdio_mount	Selecting CMD0 issuance when SDIO card is mounted
	sddev_cmd8_sdio_mount	Selecting CMD8 issuance when SDIO card is mounted

3.3 Memory for Library Use

This section describes the defining and initializing method of memory area that the library uses.

3.3.1 Library Function Work Area and Buffer Area

The work area that the library function of SD driver uses must be retained by the application. The SD driver library function operates the work area retained by the application by setting it to the initialization function of the library function.

As the work area, the area for the macro definition size shown in Table 3.5 is necessary. The work area should be assigned in 8-byte boundary.

The work area should be maintained from the initialization to the end of the driver function of SD driver. The maintaining method of the library function work area can be either dynamic or static. Moreover, the content of the library function work area directly from the user program must not be changed. The operation for the case of changing the contents of work area by user program cannot be guaranteed.

The example of defining work area is shown in Figure 3.1.

Moreover, the buffer area for data passing with the card used inside the library function should be retained by the application. The minimum buffer area is 512 bytes, and it is set in the 512 bytes unit according to the buffer area setting function. The buffer area is used as the register read at mounting and the initialization data area at formatting.

Table 3.5 Macro Definition for Library Work Area

Macro Definition	Function Outline
SD_SIZE_OF_INIT	Library function work area size (byte)

```
/* SD work area definition */  
uint32_t sd_driver_work[SD_SIZE_OF_INIT/sizeof(uint32_t)];
```

Figure 3.1 Defining Example of Library Function Work Area

3.3.2 Specification of Library Function Work and Buffer Areas

The specification of the work area for library functions is performed by the `sd_init` initialization function. Refer to the description of `sd_init` function for details. Moreover, the library buffer area is executed by the `sd_set_buffer` function. Refer to the description of `sd_set_buffer` function for details.

3.4 Status Confirmation Method

In the operation of the SD memory card, the confirmation of SD host controller status like the detection of communication end and the detection of card swapping should be executed. This section describes the status confirmation method when the library function of SD driver is used.

3.4.1 Status Confirmation Method

The SD driver supports two methods of confirming the status of the SD host controller: SD host controller interrupt (hardware interrupt) and software polling. Moreover, there are the card swapping detection and SD protocol control as the kind of status to confirm. The status confirmed by the library function of SD driver is shown in Table 3.6.

Table 3.6 Status to Confirm

Classification	Status	Remarks
Card swapping detection	Card insertion	-
	Card extraction	-
SD protocol control	Response reception completion	Generated every command transmission
	Data transfer request	Generated 512-byte or SDIO block-size transfer
	Protocol error	When CRC error occurs
	Time-out error	When it is no response

The status confirmation method of the SD protocol control is set by the `sd_mount` function. Moreover, the status confirmation method of card swapping detection is set by the library function `sd_cd_int` function. Refer to the description of each function for the details of setting method.

When interrupt is selected for the status confirmation method, it is necessary to register the library function `sd_int_handler` function to the system as an interrupt handler corresponding to the SD host controller interrupt. Moreover, the process that enables the SD host controller interrupt such as the CPU interrupt controller setting should be set inside the target CPU interface function although the interrupt setting in the SD host controller is set by the library function.

Note: When hardware interrupt is selected as the method for status confirming card swapping detection, hardware interrupt should be also selected for confirmation of SD protocol control status.

3.4.2 Card Swapping Detection

When the software polling is specified for the status confirmation method of card swapping detection, the card insertion can be confirmed by the library function `sd_check_media` function.

When the interrupt is set to the status confirmation method of card swapping detection, the card insertion can be confirmed by the library function `sd_check_media` function as well as the function specified by user can be called as the callback function when the interrupt of card swapping is generated. The function specified by user is called when the interrupt at card swapping is generated so that the real-time process to card swapping at is possible. The user specified function called when the card swapping is generated is set by the library function `sd_cd_int` function.

3.4.3 SD Protocol Control Status Confirmation by Interrupt

When hardware interrupt is selected as the method for confirming SD protocol control, the response receiving waiting time and the data transfer waiting time when communicating with the card can be allocated in the other process. The SD protocol control status confirmation method is specified by the `sd_mount` function. Moreover, when the interrupt of status confirmation is generated, the user specified function can be called as a callback function so that the flexible supports to the interrupt generation waiting process is possible. The interrupt callback function is registered by the `sd_set_intcallback` function.

Users should make the interrupt waiting processing as the target CPU interface function `sddev_int_wait` function. The flowchart example of the SD protocol control status confirmation when the interrupt is used is shown in Figure 3.2.

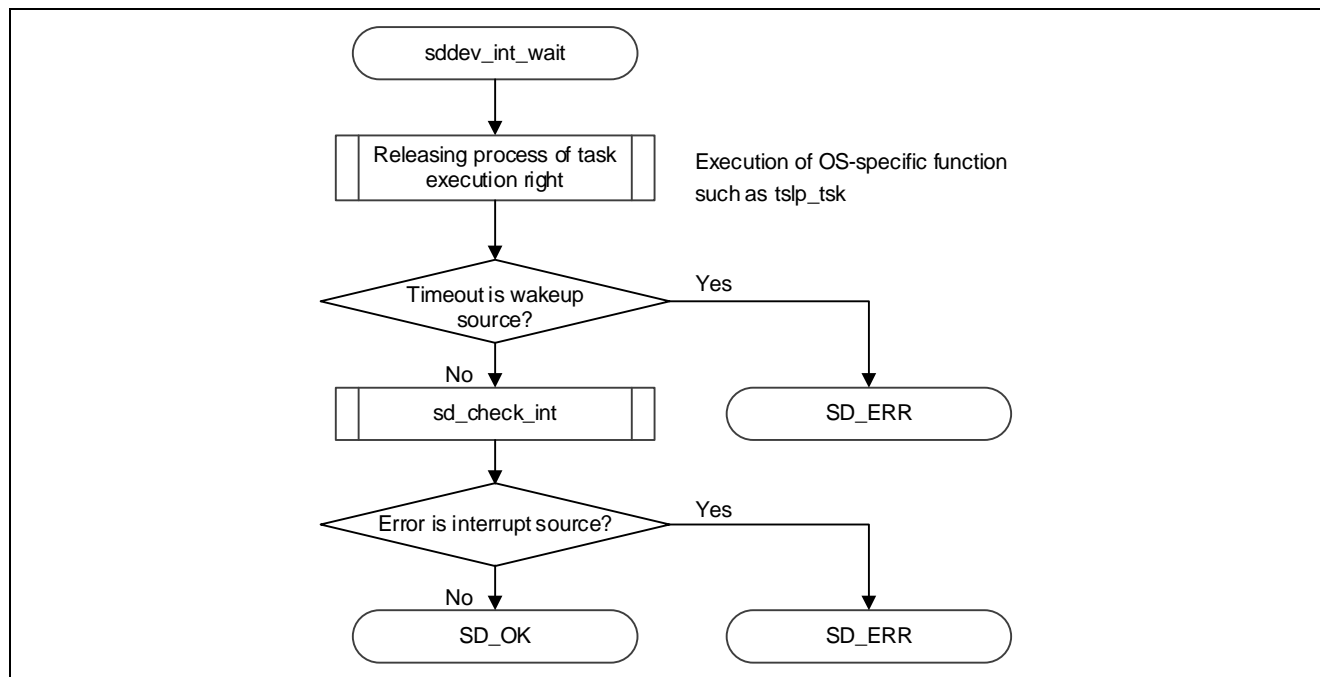


Figure 3.2 Example of Status Confirmation by Interrupt

3.4.4 SD Protocol Control Status Confirmation by Polling

When the software polling is specified for the status confirmation method of SD protocol control, the response receiving waiting when communicating with the card and the data transfer completion waiting are confirmed by the software polling. The SD protocol control status confirmation method is specified by the `sd_mount` function.

The status change is confirmed by using the library function `sd_check_int` function at the software polling. The software polling is executed in the target CPU interface function `sddev_int_wait` function. The flowchart example of the SD protocol control status confirmation when software polling is used is shown in Figure 3.3.

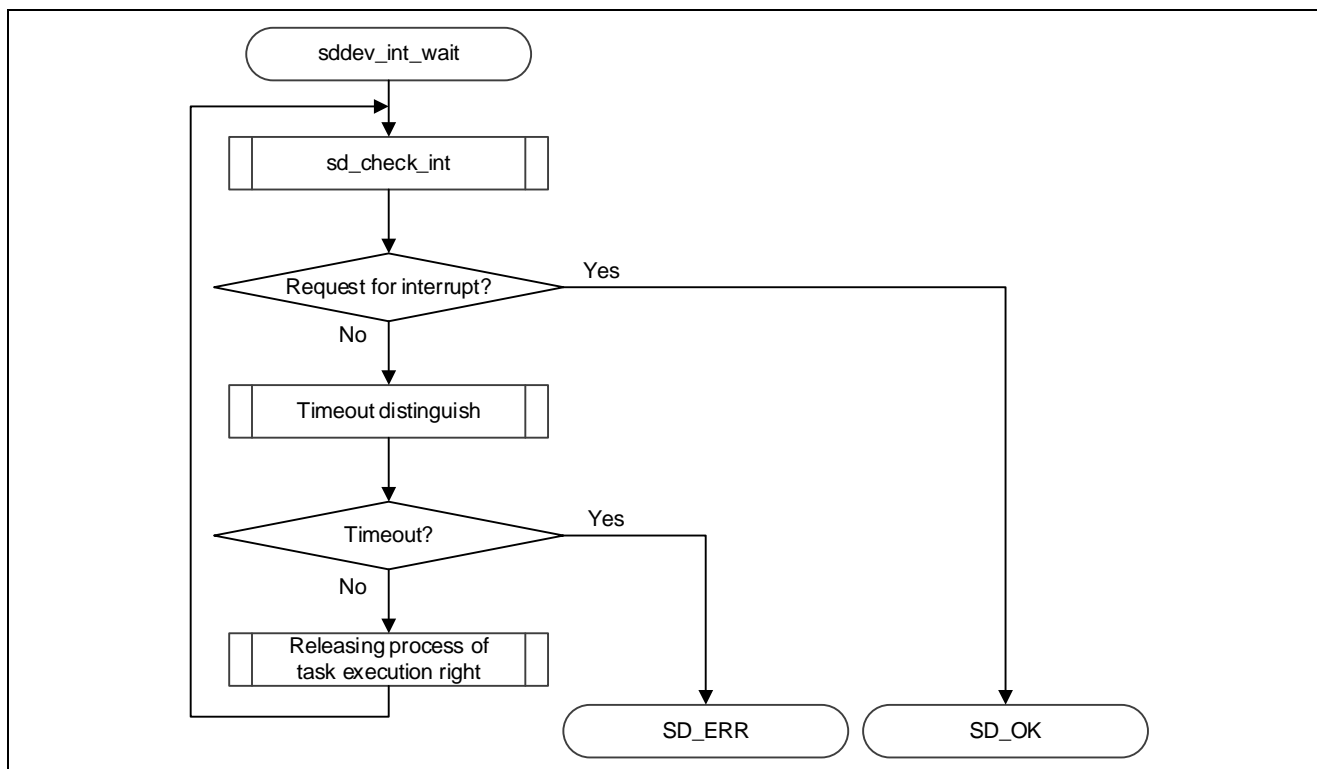


Figure 3.3 Example of Confirming Status by Software Polling

3.5 SDCLK Decision Method

The SDCLK supplied to the card divides the frequency of the clock supplied to SD host controller IP and outputs. Here, the SDCLK output from the SD host controller is described.

3.5.1 Clock Frequency Dividing Ratio

Because the clock supplied to the SD host controller is different depending on the system, the frequency dividing ratio of clock frequency that outputs as SDCLK at the target CPU interface function `sddev_get_clockdiv` function should be decided according to the system. Refer to the description of the `sddev_get_clockdiv` function for details.

The frequency of SDCLK is 400 kHz in maximum in the card identification mode, and is 25 MHz in maximum (50MHz in maximum only for SDIO cards) in the data transfer mode. However, the maximum frequency in the data transfer mode is decided from the content of the CSD register of the card at the library function, and it is specified for an argument of the `sddev_get_clockdiv` function. In the `sddev_get_clockdiv` function, the frequency dividing ratio of SDCLK should be decided not to exceed the frequency specified by the library function.

3.5.2 Stop of Clock

In the SD driver, the SDCLK is output only while the library function is executed to lower the power consumption of the card, and when the execution of the library function is finished, the SDCLK output is stopped.

3.6 Sector Data Transfer Method

As the method of the sector data transfer with the SD memory card, either of transferring with software or transferring by DMA can be selected. This section describes the method of the sector data transfer when the SD driver library function is used. The sector data is transferred when the library function `sd_read_sect` function, and the `sd_write_sect` function are used.

3.6.1 Transferring Method with Software

When the software transfer is selected as the method of transferring the sector data, the sector data is transferred by the target CPU interface function `sddev_read_data` function and the `sddev_write_data` function. The selection of the transferring method is specified by the library function `sd_mount` function.

Refer to the function description of the `sddev_read_data` function and the `sddev_write_data` function for details of software transfer method.

3.6.2 Transferring Method with DMA

When the DMA transfer is selected as the method of sector data transferring, the sector data is transferred by using the DMA controller of CPU.

When the DMA transfer is selected, the address of 8-byte boundary should be specified to the address in the buffer specified for the library function `sd_read_sect` function and the `sd_write_sect` function.

When the address in the buffer specified for the library function `sd_read_sect` function and the `sd_write_sect` function are not 8-byte boundary, the library function doesn't execute the DMA processing and executes software transfer.

The DMA controller setting and the confirmation for end are executed by the target CPU interface function `sddev_init_dma` function, the `sddev_wait_dma_end` function, and the `sddev_disable_dma` function. Refer to each function description for details of the setting.

3.7 High-Capacity Card and eXtended-Capacity Card Support

This library supports to the High-Capacity SD memory card with memory capacity that exceeds 2 GB (32 GB in max.) specified in SD Memory Card Specifications Part 1 ver. 2.00 and eXtended-Capacity SD memory card with memory capacity that exceeds 32 GB (2 TB in max) provided by SD Memory Card Specifications Part 1 ver. 3.00. However, to support the High-Capacity card, the filesystem which is the upper layer of SD driver must support FAT32. Also, to support eXtended-Capacity card, the upper layer filesystem of SD driver must support exFAT.

3.7.1 Selecting for High-Capacity Card Support and eXtended-Capacity Card Support

This library can auto-detect and mount between the eXtended-Capacity card, the High-Speed card, and the Standard-Capacity card. Moreover, it is also possible to support only the Standard-Capacity mode. The discrimination method is specified by the argument of the `sd_mount` function.

The specification and operation mode at mounting are shown in Table 3.7.

Table 3.7 Inserted Card and Operation Mode (3)

Specification at Mounting	Inserted Card		
	eXtended-Capacity Card	High-Capacity Card	Standard-Capacity Card
SD_MODE_VER2X	eXtended-Capacity mode	High-Capacity mode	Standard-Capacity mode
SD_MODE_VER1X	Error	Error	Standard-Capacity mode

3.7.2 Obtaining Operation Mode

The operation mode of the library can be obtained at the library function `sd_get_type` function. Refer to the description of the `sd_get_type` function for details.

3.8 SDIO Card Support

This library supports the SDIO card provided by the SD Specifications Part E1 SDIO Specification Ver 3.00, which enables the reading of the card common control registers (CCCR) and the function basic registers (FBR), and to execute I/O control using the I/O read/write commands.

3.8.1 Selecting SDIO Card Support

This library has two operation modes: it can auto-detect and mount an SDIO card or SD memory card, or it can alternatively it can be set to only detect and mount an SD memory card. The operation mode is specified by the argument of the `sd_mount` function. The specification and operation mode at mounting are shown in Table 3.8.

Table 3.8 Inserted Card and Operation Mode (4)

Specification at mounting	Inserted Card	
	SDIO card	SD memory card
SD_MODE_IO	SDIO card	SD memory card
SD_MODE_MEM	Error	SD memory card

3.8.2 Obtaining Operation Mode

The operation mode of the library can be obtained at the library function `sd_get_type` function. Please refer to the description of the `sd_get_type` function for details.

3.8.3 Detecting SDIO Interrupts

This library detects the SDIO interrupts provided by the SD Specifications PartE1 SDIO Specification Version 3.00. Either the SD host controller interrupt or software polling is selectable as the detection method of the SDIO interrupts. When the SD host controller interrupt is selected for the detection method, the library function `sdio_int_handler` function should be registered with the system as the interrupt handler corresponding to the SD host controller interrupt. The interrupt setting in the SD host controller should be executed by the library function, however the processing to enable the SD host controller interrupt such as interrupt controller setting in the CPU should be specified in the target CPU interface function.

3.8.4 Detecting SDIO Interrupts by SD Host Controller Interrupt

If the SD host controller interrupt is specified for the detection method of the SDIO interrupts, an interrupt is generated at the SDIO interrupt generation. This detection method is specified by the `sd_mount` function, and responds flexibly to the interrupt generation waiting process because it can call the function specified by user as a callback function when the SDIO interrupt is generated. The interrupt callback function is registered in the `sdio_set_intcallback` function.

Once the SDIO interrupt is generated, the processing to disable the interrupts is executed by the library function `sdio_int_handler` function, and the function specified by user is called as a callback function.

The method to clear the SDIO interrupt source depends on the SDIO card, but such interrupt source is generally cleared by the write processing to the registers mounted in the SDIO card. However, because this library cannot control the I/O of SDIO card in the interrupt handler processing, the SDIO interrupt generation should be notified from the SDIO interrupt callback function to the processing except for the interrupt handler, and the I/O of SDIO card should be controlled by the processing (not the interrupt handler processing) to clear the SDIO interrupt source. After the SDIO interrupt source has been cleared, the SDIO interrupt is enabled by the library function `sdio_enable_int` function.

Figure 3.4 shows the flowchart example of the SDIO interrupt handler processing.

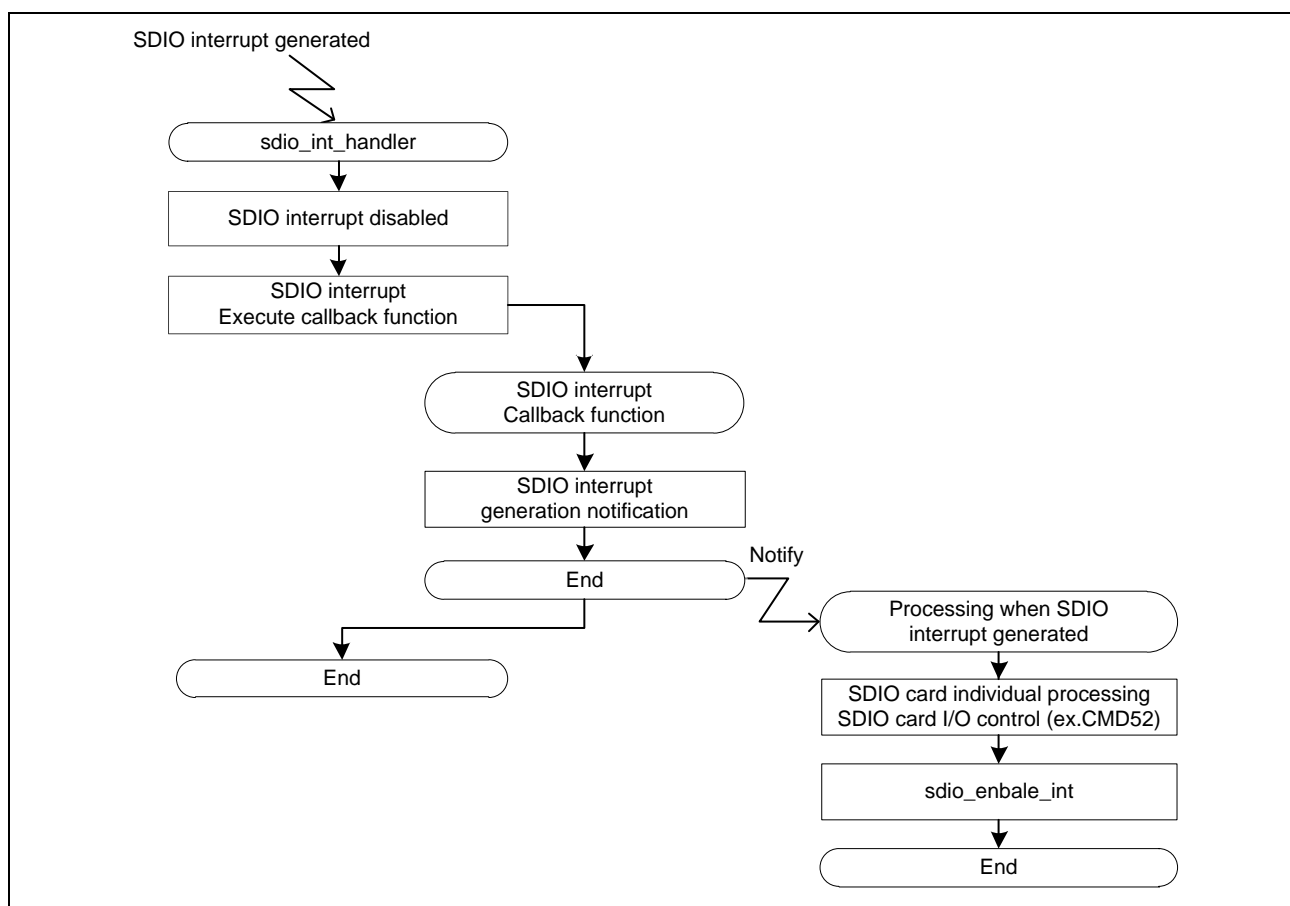


Figure 3.4 Example of SDIO Interrupt Handler Processing

3.8.5 Detecting SDIO Interrupts by Software Polling

When the software polling is specified for the detection method of the SDIO interrupts, the SDIO interrupt generation is confirmed by using the library function `sdio_check_int` function. The detection method of the SDIO interrupts is specified by the `sd_mount` function.

Figure 3.5 shows the flowchart example of the processing when the software polling is selected.

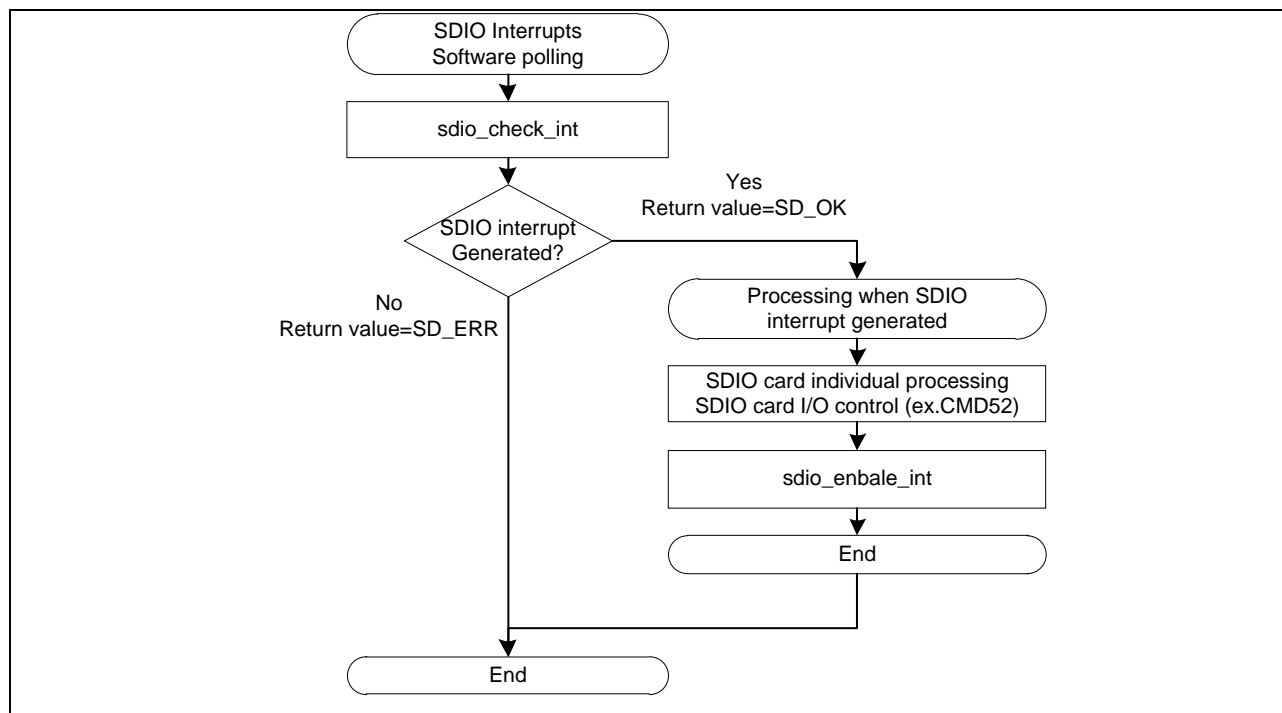


Figure 3.5 Example of SDIO Interrupt Generation Confirmation by Software Polling.

3.9 SDIO Block Data Transfer Method

As the method of the block data transfer with the SDIO card, either of transferring with software or transferring by DMA can be selected. This section describes the method of the SDIO block data transfer when the SD driver library function is used. The SDIO block data is transferred when the library functions `sdio_read` function and the `sdio_write` function are used.

3.9.1 Software Transfer Method

When the software transfer is selected as the method of transferring the SDIO block data, the data is transferred by the target CPU interface functions `sddev_read_data` function and the `sddev_write_data` function. The selection of the transferring method is specified by the library function `sd_mount` function.

Please refer to the descriptions of the `sddev_read_data` function and the `sddev_write_data` function for details about the software transfer method.

3.9.2 DMA Transfer Method

When the DMA transfer is selected as the method of SDIO block data transferring, the data is transferred by using the DMA controller of CPU.

When the DMA transfer is selected, the address of 8-byte boundary should be specified to the address in the buffer specified for the library functions `sdio_read` function and `sdio_write` function.

When the address in the buffer specified for the library functions `sdio_read` function and `sdio_write` function is not 8-byte boundary, the library function does not execute the DMA processing, but executes software transfer.

The DMA control setting and the confirmation for end are executed by the target CPU interface functions `sddev_init_dma` function, `sddev_wait_dma_end` function, and `sddev_disable_dma` function. Please refer to function descriptions for details of the setting.

3.10 Speed-up of Data Transfer Processing

Here, it describes the method of data transfer in high speed by the application using SD driver.

3.10.1 Card Format

The SD memory card has a specification to which high-speed data transfer can be processed for the recommended FAT format. Therefore, the SD card re-formatted without using special utility in PC is mostly not the best suitable FAT format; as a result, the performance is not come out.

3.10.2 Data Transfer Size (SD Memory Card)

In the SD driver, the maximum values of the number of continuous transferring sectors in the `sd_write_sect` function and the `sd_read_sect` function can be set by the `sd_set_seccnt` function. The data-transfer speed depends on this number of continuous transferring sectors, and the number of sectors specified for the `sd_write_sect` function and the `sd_read_sect` function.

The larger the number of continuous transferring sectors set to the `sd_set_seccnt` function is, the better efficiency of transferring it gets. However, when the number of continuous transferring sectors is enlarged, the response speed of the cancelling of data transfer by the `sd_stop` function might become slow in the application that needs the cancel processing of the data transfer. The multiple of 256 or more is recommended for the number of continuous transferring sectors.

In the `sd_write_sect` function or the `sd_read_sect` function, it is transferred based on this number of continuous transferring sectors. It is transferred by the specified number of sector when the number of sectors specified for the `sd_write_sect` function or the `sd_read_sect` function is below the number of continuous transferring sectors. When the number of sectors specified for the `sd_write_sect` function or the `sd_read_sect` function is larger than the number of continuous transferring sectors, it is transferred by dividing each number of continuous transferring sectors.

For the sector data writing, the data-transfer speed is greatly different depending on the writing size.

According to the characteristic of the SD memory card, writing the number of sectors that doesn't come up to the number of sectors of one cluster will be the most inefficient. It is more efficient to write by specifying a large value as much as possible for number of sectors specified for the `sd_write_sect` function though the best writing size is different according to the application.

The continuous reading is processed in each sector set by the `sd_set_seccnt` function similarly for the sector data reading. It is more efficient to specify a large value as much as possible for the number of sectors specified for the `sd_read_sect` function. However, there is no big difference in speed compared with the case of writing.

Note: The data-transfer speed is greatly different according to the SD memory card to use.

3.10.3 Data Transfer Size (SDIO Card)

In the SD driver, the maximum value of the number of continuous transferring blocks in the `sdio_write` function and the `sdio_read` function can be set by the `sdio_set_blkcnt` function. The data-transfer speed depends on this number of continuous transferring blocks, the data transfer size specified for the `sdio_write` function and the `sdio_read` function.

The larger the number of continuous transferring blocks set to the `sdio_set_blkcnt` function is, the better efficiency of transferring it gets. However, when the number of continuous transferring blocks is enlarged, the response speed of the cancelling of data transfer by the `sdio_abort` function might become slow in the application that needs the cancel processing of the data transfer. The multiple of 32 or more is recommended for the number of continuous transferring sectors. In the `sdio_write` function and the `sdio_read` function, the transfer is executed based on this number of transferring blocks. When the data transfer size specified for the `sdio_write` function and the `sdio_read` function is smaller than the SDIO block size, the SDIO byte transfer is executed in the specified data transfer size. When the data transfer size specified for the `sdio_write` function and the `sdio_read` function is larger than the SDIO block size, the SDIO block transfer is executed in the specified SDIO block size. Furthermore, when the data transfer size is larger than the size of the SDIO block multiplied by the number of continuous transferring blocks, the SDIO block transfer is executed by dividing the size into each number of continuous transferring blocks. If the SDIO data smaller than the SDIO block size is left, the SDIO byte transfer is continuously executed.

3.11 Card Mount Processing

This library can auto-detect and mount the connected card type. The mount processing is executed by the `sd_mount` function, which contents to be executed in the mode at mounting are different depending on the arguments.

When `SD_MODE_MEM` is specified by the argument, only the processing to mount the SD memory card (including the MMC card) is executed. When `SD_MODE_IO` is specified by the argument, the SDIO card mount processing is executed to detect the presence of SDIO function. Then the SD memory card mount processing is executed.

If the mount processing is executed by connecting the SDIO Combo card which has the SDIO function and the SD memory function, the SD memory card mount processing is executed after the SDIO card mount processing. However, the `GO_IDLE_STATE` (CMD0) command and the `SEND_IF_COND` (CMD8) command are required to execute the SD memory card mount processing using the SDIO Combo card. If the commands are not selected to issue by the target CPU interface function;

`sddev_cmd0_sdio_mount` (selecting CMD0 issuance when SDIO card is mounted) or `sddev_cmd8_sdio_mount` (selecting CMD8 issuance when SDIO card is mounted), the SD memory card mount processing is not executed after detecting the presence of the SDIO function.

After the SDIO card has been mounted, if the card is mounted again without unmounting the SD memory card (`sd_unmount`) or resetting the SDIO (`sdio_reset`), the mount processing should be executed after the SDIO reset processing (Re-init IO processing). Please refer to SD Specifications PartE1 SDIO Specification Version 3.00 for more details about Re-init IO.

Figure 3.6 shows the overview of mount processing in this library.

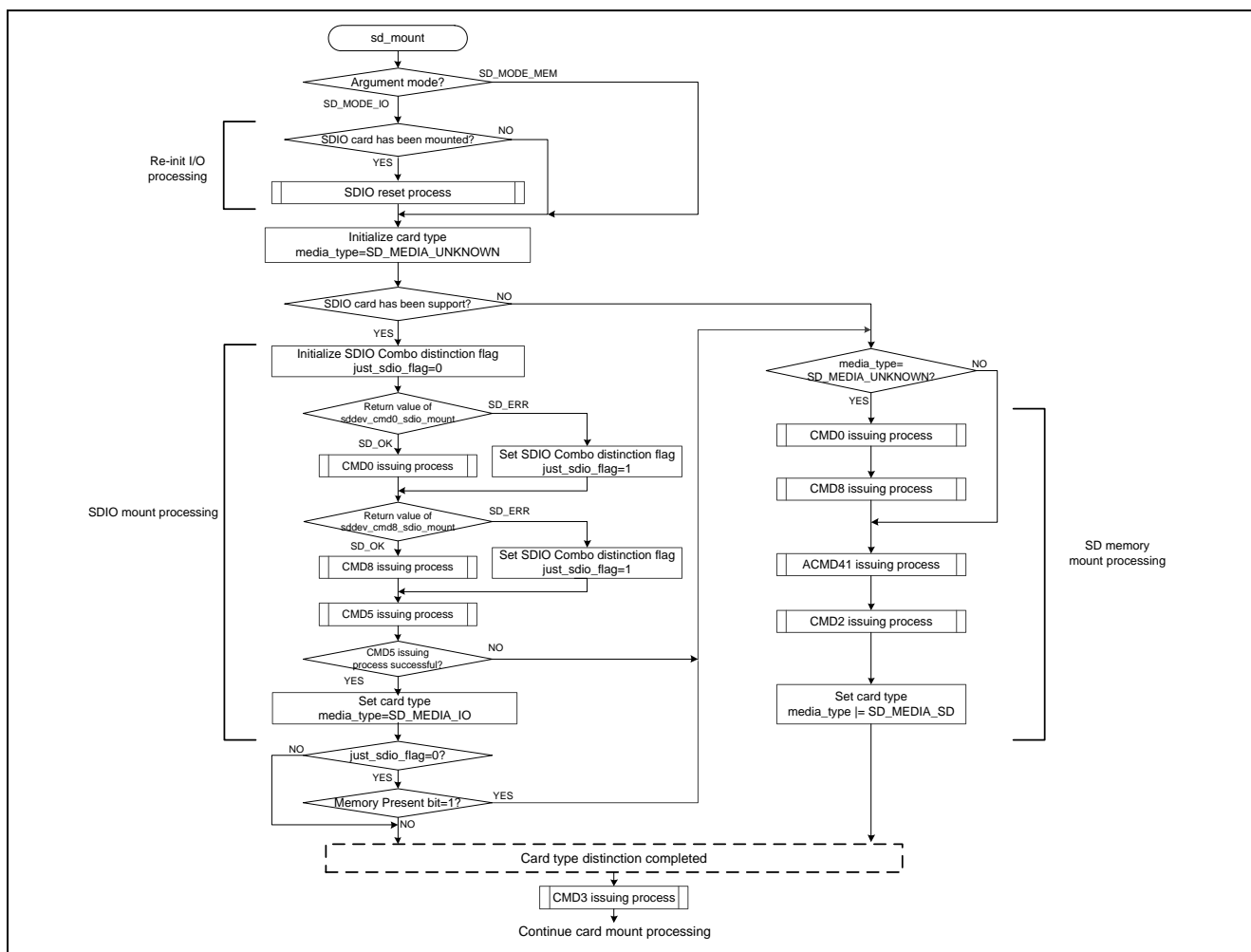


Figure 3.6 Overview of Card Mount Processing

3.12 Error Codes

When the error occurs during the processing of the library function of SD driver, it returns the error to the return value. The macro definition shown in Table 3.9 is defined as an error code of the library function. The value that doesn't exist in the table is a reservation for the future.

After the library function `sd_mount` function, the `sd_read_sect` function, and the `sd_write_sect` function are executed, the content of the error can be obtained even at the library function `sd_get_error` function. The error content of other library functions cannot be obtained at the `sd_get_error` function.

Table 3.9 Error Codes

Macro Definition	Value	Descriptions	Details
SD_OK_LOCKED_CARD	1	Normal end (SD card locked)	Locked card successfully mounted
SD_OK	0	Normal end	Normal end
SD_ERR	-1	General error	<code>sd_init</code> function is not executed, argument parameter error, etc.
SD_ERR_WP	-2	Write protect error	Writing to the write protected card
SD_ERR_RES_TOE	-4	Response time-out	The responses to the command cannot be received within 640 clocks (SDCLK).
SD_ERR_CARD_TOE	-5	Card time-out	Time out in the state of card busy Data receiving time-out after read command CRC status receiving time-out after write command
SD_ERR_END_BIT	-6	End bit error	The end bit was not able to be detected
SD_ERR_CRC	-7	CRC error	CRC error detection on host side
SD_ERR_HOST_TOE	-9	Host time-out error	Error of <code>sddev_int_wait</code> function
SD_ERR_CARD_ERASE	-10	Card erase error	SD card status error (<code>ERASE_SEQ_ERROR</code> or <code>ERASE_PARAM</code>) Erase sequence or erase command parameter error
SD_ERR_CARD_LOCK	-11	Card lock error	SD card status error (<code>CARD_IS_LOCKED</code>) Operation to locked card
SD_ERR_CARD_UNLOCK	-12	Card unlock error	SD card status error (<code>LOCK_UNLOCK_FAILED</code>) Error when the card lock is released
SD_ERR_HOST_CRC	-13	Host CRC error	SD card status error (<code>COM_CRC_ERROR</code>) CRC error detection on card side
SD_ERR_CARD_ECC	-14	Card ECC error	SD card status error (<code>CARD_ECC_FAILED</code>) ECC error generation inside the card
SD_ERR_CARD_CC	-15	Card CC error	SD card status error (<code>CC_ERROR</code>) Error in the card internal controller
SD_ERR_CARD_ERROR	-16	Card error	SD card status error (<code>ERROR</code>) Error on card side
SD_ERR_CARD_TYPE	-17	Unsupported card	Recognized as unsupported card
SD_ERR_NO_CARD	-18	Card not inserted error	The card has not been inserted.

Macro Definition	Value	Descriptions	Details
SD_ERR_ILL_READ	-19	Incorrect reading error	The method of reading the sector data by the sddev_read_data function or the DMA transfer is incorrect
SD_ERR_ILL_WRITE	-20	Incorrect writing error	The method of reading the sector data by the sddev_read_data function or the DMA transfer is incorrect
SD_ERR_AKE_SEQ	-21	Authentication process sequence error	SD card status error (AKE_SEQ_ERROR)
SD_ERR_OVERWRITE	-22	CSD overwrite error	One of the following errors: <ul style="list-style-type: none"> CSD read-only section does not match card contents. Attempt made to copy or invert permanent WP bit.
SD_ERR_CPU_IF	-30	Target CPU interface function error	Error of target CPU interface function (Excluding the sddev_int_wait function)
SD_ERR_STOP	-31	Termination error	Termination by the sd_stop function or the sdio_abort function
SD_ERR_CSD_VER	-50	CSD version error	CSD structure version has the irregular of SD Memory Card Physical Specification Version
SD_ERR_FILE_FORMAT	-52	File format error	CSD register file format error
SD_ERR_NOTSUP_CMD	-53	Unsupported command error	An unsupported command was specified.
SD_ERR_ILL_FUNC	-60	SDIO illegal function	SDIO illegal function
SD_ERR_IO_VERIFY	-61	Verification error after SDIO write	Verification error after SDIO direct write
SD_ERR_IO_CAPAB	-62	SDIO capability error	SDIO capability error
SD_ERR_IFCOND_VER	-70	Interface condition version error	The version of interface condition is incorrect
SD_ERR_IFCOND_ECHO	-72	Interface condition echo back error	The echo back pattern of interface condition is incorrect
SD_ERR_OUT_OF_RANGE	-80	Argument out of range error	SD card status error (OUT_OF_RANGE)
SD_ERR_ADDRESS_ERROR	-81	Address error	SD card status error (ADDRESS_ERROR)
SD_ERR_BLOCK_LEN_ERROR	-82	Block length error	SD card status error (BLOCK_LEN_ERROR)
SD_ERR_ILLEGAL_COMMAND	-83	Illegal command error	SD card status error (ILLEGAL_COMMAND)
SD_ERR_RESERVED_ERROR18	-84	-	-
SD_ERR_RESERVED_ERROR17	-85	-	-
SD_ERR_CMD_ERROR	-86	Command index error	SDIF internal error The transmit command index does not match the receive command index
SD_ERR_CBSY_ERROR	-87	Command error	SDIF internal error Command is busy
SD_ERR_NO_RESP_ERROR	-88	No response error	SDIF internal error Response cannot be received
SD_ERR_ERROR	-96	SDIO error	SDIO error
SD_ERR_FUNCTION_NUMBER	-97	SDIO function number error	SDIO function number error
SD_ERR_COM_CRC_ERROR	-98	SDIO CRC error	SDIO CRC error
SD_ERR_INTERNAL	-99	Internal error	Internal error of SD driver

4. Function Reference

4.1 Function Reference Details

This chapter shows the details of the SD driver library function, the target CPU interface function, and the device driver function. The details of each function are as follows.

Function Name		Type
Function outline		
Format	Describes the calling format of function. The header file indicated by #include "header file" is the standard header file necessary to execute this function. It must be included. "I" and "O" indicate that the argument is input data or output data.	
Return values	Indicates the return value of function. The comment described with ":" after the return value is the explanation of that return value such as return condition, etc.	
Description	Explains function specification	
Notes	Shows notes or cautions	
Usage example	Shows examples of use for function	
Creation example	Shows examples of creation for function	

Figure 4.1 Description of Library Function Details

4.2 Library Function

This section describes the details of library function of SD driver shown in Table 4.1.

Table 4.1 SD Driver Library Functions

Function Name	Function Outline
sd_init	Initialization of SD memory card driver
sd_finalize	Termination of SD memory card driver
sd_set_buffer	Buffer area setting for library
sd_cd_int	Card swapping interrupt setting
sd_check_media	Confirmation of card insertion
sd_set_seccnt	Continuous transfer sector count setting
sd_get_seccnt	Continuous transfer sector count acquisition
sd_mount	Card mounting
sd_unmount	Card mount releasing
sd_inactive	Card disabling
sd_read_sect	Sector read from SD card
sd_write_sect	Sector write to SD card
sd_get_type	Card type and operation mode acquisition
sd_get_size	Card size acquisition
sd_iswp	Card write-protect state acquisition
sd_stop	Forced termination of card processing
sd_set_intcallback	Registration of protocol status confirmation interrupt callback function
sd_int_handler	Card interrupt handler
sd_check_int	Card interrupt request confirmation
sd_get_reg	Card register acquisition
sd_get_rca	RCA register acquisition
sd_get_sdstatus	SD status acquisition
sd_get_error	Driver error acquisition
sd_set_cdtime	Card detection time setting
sd_set_responsetime	Response timeout time setting
sd_get_ver	Library version acquisition
sd_lock_unlock	Card locking/unlocking
sd_get_speed	Card speed acquisition

Table 4.2 SD Driver Library Functions (2)

Function Name	Function Outline
sdio_get_cia	Obtaining SDIO card CIA information
sdio_get_ioocr	Obtaining SDIO card IO_OCR information
sdio_get_ioinfo	Obtaining SDIO card IO information
sdio_reset	SDIO reset
sdio_set_enable	SDIO function enable
sdio_get_ready	SDIO ready confirmation
sdio_set_blocklen	SDIO block size setting
sdio_get_blocklen	Obtaining SDIO block size
sdio_set_int	SDIO interrupt setting
sdio_get_int	Obtaining SDIO interrupt setting status
sdio_read_direct	Direct-read from SDIO card
sdio_write_direct	Direct-write to SDIO card
sdio_read	Reading data from SDIO card
sdio_write	Writing data to SDIO card
sdio_enable_int	SDIO Interrupt enable (SDHI module)
sdio_disable_int	SDIO Interrupt disable (SDHI module)
sdio_set_intcallback	Registration of SDIO interrupt callback function
sdio_int_handler	SDIO interrupt handler
sdio_check_int	SDIO interrupt request confirmation
sdio_abort	SDIO driver termination
sdio_set_blkcnt	Continuous transfer block count setting
sdio_get_blkcnt	Continuous transfer block count acquisition

4.2.1 sd_init

sd_init

Library function

Initialization of SD memory card driver

Format	<pre>#include "r_sdif.h" int32_t sd_init(int32_t sd_port, uint32_t base, void *workarea, int32_t cd_port) int32_t sd_port SDHI channel number (0 or 1) uint32_t base Sampling clock controller base address void *workarea Work area used in access library (The area for byte supporting SD_SIZE_OF_INIT is necessary) int32_t cd_port Setting of port for swapping detection SD_CD_SOCKET: Swapping is detected with the card socket pin. SD_CD_DAT3: Swapping is detected with the CD/DAT3 pins. (Reservation)</pre>		
Return values	SD_OK	:	Normal end
	SD_ERR	:	Error end
	SD_ERR_CPU_IF	:	Target CPU interface function error
Description	<p>The SD driver and the SD host controller are initialized.</p> <p>The base address of sampling clock controller is specified according to the argument base. In the library function, it accesses the SD host controller based on the base address. If the address is not the base address of the sampling clock controller, the function terminates with an error.</p> <p>The areas of SD_SIZE_OF_INIT (byte) specified with the argument workarea are used as the work area of the library function. The area specified with workarea should be maintained until the SD driver processing is complete and that content should not be changed by the application. Moreover, the address of 8-byte boundary should be specified for workarea. If the value of workarea is NULL or not an address aligned with an 8-byte boundary, the function terminates with an error.</p> <p>The port form used for swapping detection is specified by the argument cd_port. When SD_CD_SOCKET is specified, the CD pin of the SD card socket is used for swapping detection.</p> <p>When SD_CD_DAT3 is specified, the CD/DAT3 pin of the SD card is used for swapping detection.</p> <p>When both the SD memory card and MMC card are supported, the CD pin of the socket should be used for swapping detection. When other values are specified for the argument cd_port, it ends in error.</p> <p>After this function is executed, the swapping detection interrupt is disabled. When the swapping interrupt is used, the swapping interrupt should be enabled at the sd_cd_int function.</p> <p>The target CPU interface function sddev_init function is called from this function. The hardware other than the SD host controller such as port setting and interrupt setting necessary for the card operation should be initialized in the sddev_init function.</p>		
Notes	<p>When this function doesn't end normally, all the library functions cannot be used.</p> <p>In this function, the power is not supplied to the card.</p> <p>Setting of the port for swap detection is optional. This setting value is valid only when the SD</p>		

card detection option is enabled. For details, refer to the SD card detection option in Chapter 5, Configuration Options.

The setting of port SD_CD_DAT3 for swapping detection is a reservation for the future.

When SD_CD_DAT3 is specified in this version, the same operation as SD_CD_SOCKET is executed.

The error by the sd_get_error function cannot be obtained.

Usage example

```
/* Example from driver initialization to termination */
#include "r_sdif.h"

/* SD driver work area definition */
uint32_t driver_work[SD_SIZE_OF_INIT/sizeof(uint32_t)];

/* SD driver buffer area definition */
uint32_t my_sd_buffer[512/sizeof(uint32_t)];

/* Sampling clock controller base address is set */
#define HOST_IP_ADDR    (0xE8227000uL)

void func(void)
{
    /* Driver initialization */
    if(sd_init(0, HOST_IP_ADDR, driver_work, SD_CD_SOCKET) != SD_OK)
    {
        /* Initialization is failed */
    }

    /* Buffer setting for SD driver */
    sd_set_buffer(0, my_sd_buffer, sizeof(my_sd_buffer));

    /* Mount with software polling, DMA transfer, default-Speed card
       supporting, High-Capacity card supporting, and eXtended-Capacity
       card supporting */
    if(sd_mount(0
                ,SD_MODE_POLL|SD_MODE_DMA|SD_MODE_DS|SD_MODE_VER2X
                ,SD_VOLT_3_3) != SD_OK)
    {
        /* Mount is failed */
    }

    /* Access processing to the card */

    /* Releasing mount */
    sd_unmount(0);

    /* Driver termination */
    sd_finalize(0);
}
```

4.2.2 sd_finalize

sd_finalize

Library function

Termination of SD memory card driver

Format `#include "r_sdif.h"`
 `int32_t sd_finalize(int32_t sd_port)`
 `int32_t sd_port` `I` SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description All the processing of SD memory card driver is ended.
 After this function is executed, the swapping interrupt is also disabled.
 The work area of the SD driver that is set to the `sd_init` function should be released after
 executing this function.
 The target CPU interface function `sddev_finalize` function is called from this function. In the
 `sddev_finalize` function, the hardware termination processing other than the SD host
 controllers such as the port setting or the interrupt setting used in the SD memory card
 operation should be executed.

Notes When this function is executed before calling of the `sd_init` function, it ends in error.

Usage For details, refer to the using example of `sd_init` function.
example

4.2.3 sd_set_buffer

sd_set_buffer

Library function

Buffer area setting for library

Format `#include "r_sdif.h"`

`int32_t sd_set_buffer(int32_t sd_port, void *buff, uint32_t size)`

`int32_t sd_port` | SDHI channel number (0 or 1)

`void *buff` | Pointer to library buffer area

`uint32_t size` | Buffer area size

Return `SD_OK` : Normal end

values `SD_ERR` : Error end

Description The buffer area used in the library function is set.

The first address in the buffer area is specified for the argument `buff` and the size of the buffer area is specified for `size`. When 0 is specified for the size, it ends in error.

The area of the multiple in 512 bytes arranged in 8-byte boundary should be specified for buffer area `buff`. When the pointer that is not arranged in 8-byte boundary is specified for `buff`, it ends in error.

At least 512 bytes is necessary for the buffer area.

Notes The buffer area should be set before executing the `sd_mount` function.

Usage For details, refer to the using example of `sd_init` function.
example

4.2.4 sd_cd_int

sd_cd_int

Library function

Card swapping interrupt setting

Format	#include "r_sdif.h"	
	int32_t sd_cd_int(int32_t sd_port, int32_t enable, int32_t (*callback)(int32_t, int32_t))	
	int32_t sd_port	I SDHI channel number (0 or 1)
	int32_t enable	I Swapping interrupt disable and enable setting
		SD_CD_INT_ENABLE: Swapping interrupt enabled
		SD_CD_INT_DISABLE: Swapping interrupt disabled
	int32_t (*callback)	I Callback function specification for swapping detection
	(int32_t, int32_t)	

Return values	SD_OK	: Normal end
	SD_ERR	: Error end

Description Interrupt for the card swapping detection is set.

When SD_CD_INT_ENABLE is set to the disable and enable of swapping interrupt setting enable, the swapping interrupt is enabled. When the swapping interrupt is enabled, the library function sd_int_handler function is set to the system as a processing routine of the SD host controller interrupt, and it is necessary to enable the SD host controller interrupt by the interrupt controller of CPU. The interrupt controller of CPU should be set in the sddev_init function. Moreover, when the swapping interrupt is enabled, the user processing can be executed for the events of swapping by registering callback function for swapping detection to the argument callback. When a null pointer is specified for the argument callback, the callback function is not registered. Refer to the description of sd_cd_callback function for details of the callback function.

When SD_CD_INT_DISABLE is set to the disable and enable of swapping interrupt setting enable, the swapping interrupt is disabled. The swapping of the card by the sd_check_media function should be confirmed when the swapping interrupt is disabled.

When the values other than SD_CD_INT_ENABLE or SD_CD_INT_DISABLE are set to the argument enable, it causes an error.

Notes The hardware interrupt SD_MODE_HWINT must be specified for the status confirming method of operation mode set by the sd_mount function when SD_CD_INT_ENABLE is set to the disable and enable of swapping interrupt setting enable.

The error by the sd_get_error function cannot be obtained.

The swapping interrupt is generated by swapping the card after executing this function.

This function is optional. It can only be used when the SD card detection option is enabled.

Usage example

```
/* Card swapping interrupt setting example */
#include "r_sdif.h"

int32_t cd_callback(int32_t sd_port, int32_t in)
{
    /* Process supporting swapping */
}

void func(void)
{
    /* Swapping interrupt enabled and the callback function are
       registered */
    sd_cd_int(0, SD_CD_INT_ENABLE, cd_callback);
}
```

4.2.5 sd_check_media**sd_check_media**

Library function

Confirmation of card insertion

Format `#include "r_sdif.h"`
 `int32_t sd_check_media(int32_t sd_port)`
 `int32_t sd_port` `I` SDHI channel number (0 or 1)

Return `SD_OK` : Card inserted
 values `SD_ERR` : Card not inserted

Description The SD memory card insertion is confirmed.
 When the SD memory card has been inserted, `SD_OK` is returned.
 When the SD memory card has not been inserted, `SD_ERR` is returned.
 This function can be used when the detection by the interrupt is selected as the swapping
 detection.
 When the SD card detection option is disabled, `SD_OK` (fixed value) is returned.

Notes It is necessary to initialize this function by the `sd_init` function before it is executed.
 The error by the `sd_get_error` function cannot be obtained.

Usage `/* Card insertion checking example */`
 example `#include <stdio.h>`
 `#include "r_sdif.h"`

 `void func(void)`
 `{`
 `if(sd_check_media(0) == SD_OK)`
 `{`
 `printf("Card has been inserted\n");`
 `}`
 `else`
 `{`
 `printf("Card has not been inserted\n");`
 `}`
 `}`

4.2.6 sd_set_seccnt**sd_set_seccnt**

Library function

Continuous transfer sector count setting

Format

```
#include "r_sdif.h"
int32_t sd_set_seccnt(int32_t sd_port, int16_t sectors)
    int32_t sd_port          | SDHI channel number (0 or 1)
    int16_t sectors          | Number of continuous transferring sectors (3 to 0x7fff)
```

Return values

```
SD_OK      : Normal end
SD_ERR     : Error end
```

Description

The maximum values of the number of continuous transferring sectors to the card are set. In the `sd_write_sect` function or the `sd_read_sect` function, it is transferred based on the number of continuous transferring sectors set by this function. When the number of sectors specified for the `sd_write_sect` function or the `sd_read_sect` function is below the number of continuous transferring sectors, it is transferred by the specified number of sectors. When the number of sectors specified for the `sd_write_sect` function or the `sd_read_sect` function is larger than the number of continuous transferring sectors, it is transferred by dividing each number of continuous transferring sectors.

The initial value of the number of continuous transferring sectors is 256 sectors. When the number of continuous sectors is not set according to this function, the initial value is assumed to be a number of continuous transferring sectors. When the `sd_init` function is executed, the initial value is set.

The number of continuous transferring sectors that can be specified for the argument sectors is 3 to 0x7fff sectors. The `SD_ERR` is returned when the value is other than this value is specified.

Notes

It is necessary to initialize this function by the `sd_init` function before it is executed. The error by the `sd_get_error` function cannot be obtained.

Usage example

```
/* The number of continuous transferring sectors setting example*/
#include <stdio.h>
#include "r_sdif.h"

void func(void)
{
    if(sd_set_seccnt(0, 1024) == SD_OK)
    {
        printf("The number of continuous transferring sectors is set to
               1024 sectors\n");
    }
}
```

4.2.7 sd_get_secnt**sd_get_secnt**

Library function

Continuous transfer sector count acquisition

Format	<pre>#include "r_sdif.h" int32_t sd_get_seccnt(int32_t sd_port) int32_t sd_port SDHI channel number (0 or 1)</pre>		
Return values	≥ 1	:	Number of continuous transferring sectors
	SD_ERR	:	Error end
Description	The number of continuous transferring sectors to the card is returned. When this function is executed before the initialization by the sd_init function, SD_ERR is returned.		
Notes	It is necessary to initialize this function by the sd_init function before it is executed. The error by the sd_get_error function cannot be obtained.		
Usage example	<pre>/* Continuous transfer sector count acquisition example */ #include <stdio.h> #include "r_sdif.h" void func(void) { int32_t secCnt; secCnt = sd_get_seccnt(0); printf("The number of continuous transferring sectors is %d sectors\n", secCnt); }</pre>		

4.2.8 sd_mount

sd_mount

Library function

Card mounting

Format	#include "r_sdif.h"		
	int32_t sd_mount(int32_t sd_port, uint32_t mode, uint32_t voltage)		
	int32_t sd_port		SDHI channel number (0 or 1)
	uint32_t mode		Operation mode
	uint32_t voltage		Card operation voltage
Return values	SD_OK_LOCKED_CARD	:	Normal end (SD card locked)
	SD_OK	:	Normal end
	Excluding the above	:	Error end (Refer to 3.12, Error Codes for details)

Description Mounts the SD card.
 Normal end of this function is followed by the card state transition to Transfer State, and sector read/write access gets enabled.
 For the normal end of mounting the locked SD card, the return value is SD_OK_LOCKED_CARD.
 The locked SD card accepts only specific commands. Therefore read/write access in the sector is disabled. To enable read/write access, mount the SD card after unlocking by the lock/unlock function.
 For the commands accepted by the locked SD card, refer to SD PHYSICAL LAYER SPECIFICATION.

The operation mode of the library function is specified for the argument mode. The operation mode is specified by the macro definition shown in Table 4.2. The operation mode is specified by the logical sum for each type shown in Table 4.2. When the SD_MODE_HWINT is set to the status confirming method, it is necessary to register the library function sd_int_handler function to the system as the processing routine of the SD host controller interrupt, and to make the SD host controller interrupt enabled by the interrupt controller of CPU. Please set the interrupt controller of CPU should be set in the sddev_init function.

The range of the voltage supplied to the SD memory card is specified for the argument voltage. The range of voltage is defined by the macro definition listed in Table 4.3. The card that cannot be operated with the specified voltage is not mounted.

The distinction between the SD memory card and the MMC card is executed in this function. The pull-up of the CD/DAT3 pin in the SD memory card is disabled when the SD memory card is recognized.

Notes When the SD_CD_INT_ENABLE is set to the enable and disable setting enable of swapping interrupt of the sd_cd_int function, the hardware interrupt SD_MODE_HWINT must be specified for the status confirming method of the operation mode set by this function.
 When the DMA transfer is used for the data access method, the buffer pointer specified for sd_read_sect/sd_write_sect should be the 8-byte boundary address.
 When this function is executed before the sd_init function and sd_set_buffer are executed, it ends in error.
When it is error, retrying is recommended to be processed in the calling of this function.

```
Usage
example    /* Card mounting example */
           #include "r_sdif.h"

           void func(void)
           {
               /* Mount with software polling, software transfer, default-Speed card
                supporting, High-Capacity card supporting, and eXtended-Capacity
                card supporting */
               if(sd_mount(0
                           ,SD_MODE_POLL|SD_MODE_SW|SD_MODE_DS|SD_MODE_VER2X
                           ,SD_VOLT_3_3) != SD_OK)
               {
                   /* Mount is failed */
               }
           }
           void func2(void)
           {
               int32_t ret;
               char_t pwd[2];

               ret=sd_mount(0
                           ,SD_MODE_POLL|SD_MODE_SW|SD_MODE_DS|SD_MODE_VER2X
                           ,SD_VOLT_3_3);
               if(ret==SD_OK)
               {
                   /* Mounting succeeds */
               }
               else if(ret==SD_OK_LOCKED_CARD)
               {
                   /* Mounting succeeds with the SD card locked and unlock the
                    password by '12' */
                   pwd[0] = 0x31;
                   pwd[1] = 0x32;
                   ret=sd_lock_unlock(0, 0x00, pwd, sizeof(pwd));
                   if(ret==SD_OK)
                   {
                       /* Unlock succeeds and re-mounts */
                       ret=sd_mount(0
                                   ,SD_MODE_POLL|SD_MODE_SW|SD_MODE_DS|SD_MODE_VER2X
                                   ,SD_VOLT_3_3);
                       if(ret==SD_OK)
                       {
                           /* Mounting succeeds */
                       }
                       else
                       {
                           /* Mount is failed */
                       }
                   }
                   else
                   {
                       /* Unlocking fails */
                   }
               }
               else
               {
                   /* Mount is failed */
               }
           }
```

}

Table 4.3 SD Driver Operating Mode

Type	Macro Definition	Value	Definition
Status confirmation method	SD_MODE_POLL	0x00000000	Software polling
	SD_MODE_HWINT	0x00000001	Hardware interrupt
Data access method	SD_MODE_SW	0x00000000	Software transfer
	SD_MODE_DMA	0x00000002	DMA transfer
Card speed supporting method	SD_MODE_DS	0x00000000	Supporting Default-Speed card
	SD_MODE_HS	0x00000040	Supporting High-Speed card (Note: Supports it only for SDIO cards)
Card capacity supporting method	SD_MODE_VER1X	0x00000000	Supporting only Standard-Capacity card
	SD_MODE_VER2X	0x00000080	Supporting High-Capacity card and eXtended-Capacity card
SDIO card supporting method	SD_MODE_IO	0x00000010	Supporting SDIO card

Table 4.4 Operation Voltage Setting

Operation Voltage (V)	Macro Definition	Value
1.6-1.7	SD_VOLT_1_7	0x00000010
1.7-1.8	SD_VOLT_1_8	0x00000020
1.8-1.9	SD_VOLT_1_9	0x00000040
1.9-2.0	SD_VOLT_2_0	0x00000080
2.0-2.1	SD_VOLT_2_1	0x00000100
2.1-2.2	SD_VOLT_2_2	0x00000200
2.2-2.3	SD_VOLT_2_3	0x00000400
2.3-2.4	SD_VOLT_2_4	0x00000800
2.4-2.5	SD_VOLT_2_5	0x00001000
2.5-2.6	SD_VOLT_2_6	0x00002000
2.6-2.7	SD_VOLT_2_7	0x00004000
2.7-2.8	SD_VOLT_2_8	0x00008000
2.8-2.9	SD_VOLT_2_9	0x00010000
2.9-3.0	SD_VOLT_3_0	0x00020000
3.0-3.1	SD_VOLT_3_1	0x00040000
3.1-3.2	SD_VOLT_3_2	0x00080000
3.2-3.3	SD_VOLT_3_3	0x00100000
3.3-3.4	SD_VOLT_3_4	0x00200000
3.4-3.5	SD_VOLT_3_5	0x00400000
3.5-3.6	SD_VOLT_3_6	0x00800000

4.2.9 sd_unmount

sd_unmount

Library function

Card mount releasing

Format `#include "r_sdif.h"`
 `int32_t sd_unmount(int32_t sd_port)`
 `int32_t sd_port` `I` SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end
 `SD_ERR_CPU_IF` : Target CPU interface function error

Description The mount of the card should be released to be removable.
 The interrupt callback function for the card swapping interrupt and card swapping
 confirmation are enabled even when this function is executed and the mount of the card is
 released.

Notes

Usage `/* Card unmounting example */`
example `#include "r_sdif.h"`

 `void func(void)`
 `{`
 `/* Unmount card */`
 `sd_unmount(0);`
 `}`

4.2.10 sd_inactive

sd_inactive

Library function

Card disabling

Format `#include "r_sdif.h"`
 `int32_t sd_inactive(int32_t sd_port)`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values Excluding `SD_OK` : Error end (Refer to 3.12, Error Codes for details)

Description The card should be transited to the inactive state from the desired state to be disabled.

Notes The card swapping is necessary to mount the disabled card again.

Usage
example

```
/* Card disabling example */
#include <stdio.h>
#include "r_sdif.h"

void func(void)
{
    /* Card is disabled */
    if(sd_inactive(0) != SD_OK)
    {
        printf("Disabling is failed\n");
    }
}
```

4.2.11 sd_read_sect**sd_read_sect**

Library function

Sector read from SD card

Format `#include "r_sdif.h"``int32_t sd_read_sect(int32_t sd_port, uint8_t *buff, uint32_t psn, int32_t cnt)`

<code>int32_t sd_port</code>	I	SDHI channel number (0 or 1)
<code>uint8_t *buff</code>	O	Pointer that indicates read-in area of sector data
<code>uint32_t psn</code>	I	Read-out starting physical sector number
<code>int32_t cnt</code>	I	Number of read-out sectors

Return values

<code>SD_OK</code>	: Normal end
Excluding <code>SD_OK</code>	: Error end (Refer to 3.12, Error Codes for details)

Description The sector data is read out from the card.
 The data for the cnt sector is read out from the sector specified by the physical sector number psn, and it is stored in the area that the argument buff shows.
 The sector data is read with the following commands;
 2 sector or below: READ_SINGLE_BLOCK command (CMD17)
 3 sector or above: READ_MULTIPLE_BLOCK command (CMD18)

If NULL is specified for the argument buff, the function terminates with an error.
 When the card is extracted while this function is being executed, the processing is cancelled and it ends in error. (Note that it applies only when the SD card detection option is enabled.)

Notes The physical sector number is specified for this function. When the sector number is specified with the logical sector number from filesystem, the logical sector should be converted to the physical sector.
 When it ends in error, retrying is recommended to be processed in the calling of this function.

Usage example

```

/* Mounting example to device driver function */
#include "r_sdif.h"
int32_t offset ; /* Offset for conversion to physical sector */

int32_t read_sector(int32_t side, uint8_t *buff, uint32_t secno
                  ,int32_t cnt)
{
    int32_t i;
    uint32_t psn;

    /* Conversion to physical sector number */
    psn = secno + offset;

    /* Three-time retry processing */
    for(i=0; i < 3; i++)
    {
        if(sd_read_sect(0, buff, psn, cnt) == SD_OK)
        {
            return 0;    /* Normal end */
        }
    }
    return -1; /* Error end */
}

```

4.2.12 sd_write_sect**sd_write_sect**

Library function

Sector write to SD card

Format	<pre>#include "r_sdif.h" int32_t sd_write_sect(int32_t sd_port, uint8_t *buff, uint32_t psn, int32_t cnt, int32_t writemode)</pre> <div> <div>int32_t sd_port</div> <div>uint8_t *buff</div> <div>uint32_t psn</div> <div>int32_t cnt</div> <div>int32_t writemode</div> </div> <div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> </div> <div> <div>SDHI channel number (0 or 1)</div> <div>Pointer that indicates write-in sector data area</div> <div>Write-in starting physical sector number</div> <div>Number of write-in sector</div> <div>Writing mode</div> </div> <div> <div>SD_WRITE_WITH_PREERASE: Write in with pre-erase</div> <div>SD_WRITE_OVERWRITE: Overwritten mode</div> </div>		
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	<p>The sector data is written in the card.</p> <p>The data of the area that the argument buff shows is written for cnt sector in the sector specified with the physical sector number psn.</p> <p>When the SD_WRITE_WITH_PREERASE is specified for the writing mode at the SD memory card, it is written after the sector for writing is deleted. Even if the SD_WRITE_WITH_PREERASE is specified for the MMC card, the same operation as the SD_WRITE_OVERWRITE is executed.</p> <p>The sector data is written with the following commands;</p> <p>2 sector or below: WRITE_BLOCK command (CMD24)</p> <p>3 sector or above: WRITE_MULTIPLE_BLOCK command (CMD25)</p> <p>If NULL is specified for the argument buff, the function terminates with an error.</p> <p>When this function is executed to the card with write-disabled, it causes an error.</p> <p>When the card is extracted while this function is being executed, the processing is cancelled. (Note that is applies only when the SD card detection option is enabled.)</p>		
Notes	<p>The physical sector number is specified for this function. When the sector number is specified with the logical sector number from filesystem, the logical sector should be converted to the physical sector.</p> <p>When the card is extracted while it is being written at specifying SD_WRITE_WITH_PREERASE, it is more likely to lose the card content compared with the case that SD_WRITE_OVERWRITE is specified.</p> <p>When it ends in error, retrying is recommended to be processed in the calling of this function.</p>		

**Usage
example**

```
/* Mounting example to device driver function */
#include "r_sdif.h"
int32_t write_sector(int32_t side, uint8_t *buff, uint32_t secno
                    ,int32_t cnt)
{
    int32_t i;
    /* Three-time retry processing */
    for(i=0; i<3 ; i++)
    {
        /* Writing example in overwritten mode */
        if(sd_write_sect(0, buff, secno, cnt, SD_WRITE_OVERWRITE)== SD_OK)
        {
            return 0;    /* Normal end */
        }
    }
    return -1;    /* Error */
}
```


4.2.13 sd_get_type

sd_get_type

Library function

Card type and operation mode acquisition

Format

```
#include "r_sdif.h"
int32_t sd_get_type(int32_t sd_port, uint16_t *type, uint16_t *speed, uint8_t *capa)
    int32_t sd_port          I   SDHI channel number (0 or 1)
    uint16_t *type           O   Pointer that indicates the card type storage area
    uint16_t *speed          O   Pointer that indicates the card speed mode storage area
    uint8_t *capa            O   Pointer that indicates the storage area of card capacity
                                type
```

Return values

SD_OK	: Normal end
SD_ERR	: Error end

Description

The card type, card speed mode and card capacity type, which are mounted, should be respectively stored in the area that type, speed and capa show.

The following values are stored in the type as the card type mounted.

SD_MEDIA_UNKNOWN
SD_MEDIA_MMC
SD_MEDIA_SD

The card speed mode mounted should be stored. 0 fixed.
When the card mounted is the MMC card, 0 is stored.

b15	14	13	12	11	10	9	8
SPT							SPT
7	6	5	4	3	2	1	b0
CUR							CUR

Bits 8 (SPT bits) shows the speed mode supported by the card mounted. Bits 0 (CUR bits) shows the speed mode when the library accesses the card.

- When the value of the SPT bits is 0b0, a card with Default-Speed mode support is mounted.
- When the value of the CUR bits is 0b0, the library accesses the card in Default-Speed mode.

Bits 1 to 7 and bits 9 to 15 are the reservation bits.

When the card mounted is the High-Capacity card or the eXtended-Capacity card, 1 is stored in the capa. When the card mounted is the Standard-Capacity card, 0 is stored in the capa.

Notes

When the argument is null pointer, that information is not stored.

**Usage
example**

```
/* Card type and operating mode acquisition example */
#include <stdio.h>
#include "r_sdif.h"

void func(void)
{
    uint8_t  capa;

    /* Acquisition of card capacity type */
    sd_get_type(0, NULL, NULL, &capa);

    if(capa == 1)
    {
        printf("The library mounts the High-Capacity card or the
               eXtended-Capacity card\n");
    }
    else
    {
        printf("The library mounts the Standard-Capacity card\n");
    }
}
```

4.2.14 sd_get_size**sd_get_size**

Library function

Card size acquisition

Format	<pre>#include "r_sdif.h" int32_t sd_get_size(int32_t sd_port, uint32_t *user, uint32_t *protect) int32_t sd_port I SDHI channel number (0 or 1) uint32_t *user O Pointer that indicates the user area size storage destination uint32_t *protect O Pointer that indicates the protect area size storage destination</pre>	
Return values	SD_OK	: Normal end
	SD_ERR	: Error end
Description	<p>The physical capacity of user area and protect area of card are stored in the area that user and protect indicate by the <u>number of sectors</u>. The number of sector x 512 is the number of bytes for the area.</p> <p>When user or protect is NULL, the size is not stored. The size of the protecting area is always 0 for the MMC card.</p>	
Notes	The size of a logical layer should be obtained from the filesystem information (the master boot record and partition boot record).	
Usage example	<pre>/* Obtaining example of card size */ #include <stdio.h> #include "r_sdif.h" void func(void) { uint32_t size,bytes; /* Obtaining card size */ sd_get_size(0, &size, NULL); bytes = size * 512; printf("Card size is %d byte and %d sector\n", bytes, size); }</pre>	

4.2.15 sd_iswp**sd_iswp**

Library function

Card write-protect state acquisition

Format	<pre>#include "r_sdif.h" int32_t sd_iswp(int32_t sd_port) int32_t sd_port SDHI channel number (0 or 1)</pre>		
Return values	SD_WP_OFF	:	No write-protect
	SD_WP_HW	:	Hardware write-protect (optional)
	SD_WP_TEMP	:	CSD register TMP_WRITE_PROTECT bit ON
	SD_WP_PERM	:	CSD register PERM_WRITE_PROTECT bit ON
	SD_WP_ROM	:	SD-ROM
Description	<p>The write-protect state of card is returned.</p> <p>The return value when the card is not mounted (When the sd_mount function is not executed) is undefined.</p>		
Notes	<p>Hardware write protection is optional. It can only be used when the write protection signal detection option is enabled. For details, refer to the SD card detection option in Chapter 5, Configuration Options.</p>		
Usage example	<pre>/* Write protection state acquisition example */ #include <stdio.h> #include "r_sdif.h" void func(void) { int32_t wp; /* Obtaining the write-in disabled state */ wp = sd_iswp(0); if(wp == SD_WP_OFF) { printf("Writing card is enabled\n"); } else { printf("Writing card is disabled\n"); } }</pre>		

4.2.16 sd_stop**sd_stop**

Library function

Forced termination of card processing

Format	<pre>#include "r_sdif.h" void sd_stop(int32_t sd_port) int32_t sd_port SDHI channel number (0 or 1)</pre>
Return values	None
Description	<p>Forcibly terminates sector read/write processing of the card.</p> <p>It is used when processing is halted by an application or extraction of the card is detected due to an interrupt or the like.</p> <p>The forcible termination of processing caused by executing this function is effective until the <code>sd_read_sect</code> function, <code>sd_write_sect</code> function, or <code>sd_mount</code> function is executed. If the <code>sd_read_sect</code> function, or <code>sd_write_sect</code> function is called after this function is executed, it ends in an error without performing any processing.</p> <p>If this function is executed while the <code>sd_read_sect</code> function, or <code>sd_write_sect</code> function is running, transfer processing halts midway.</p>
Notes	The integrity of the data on the card cannot be guaranteed if forcible termination occurs during write processing.
Usage example	<pre>/* Card processing forced termination example */ #include "r_sdif.h" void func(void) { sd_stop(0); if(sd_read_sect(0, buffer, 0, 1) != SD_OK) { /* It becomes SD_ERR_STOP error */ } }</pre>

4.2.17 sd_set_intcallback**sd_set_intcallback**

Library function

Registration of protocol status confirmation interrupt callback function

Format	<pre>#include "r_sdif.h" int32_t sd_set_intcallback(int32_t sd_port, int32_t (*callback)(int32_t, int32_t)); int32_t sd_port SDHI channel number (0 or 1) int32_t Registered callback function (*callback)(int32_t, int32_t)</pre>
Return values	<pre>SD_OK : Normal end SD_ERR : Error end</pre>
Description	<p>The interrupt callback function for the SD memory card protocol status confirmation is registered.</p> <p>When interrupt is generated when the protocol status of the SD memory card changes, the callback function registered by this function is called.</p> <p>In the registered callback function, the process such as releasing the waiting state of task doing the interrupt generation waiting is executed.</p> <p>When the callback function is used, the callback function should be registered before the sd_mount function is executed.</p> <p>When the callback function is not defined, the callback function is not called by the interrupt processing.</p> <p>Moreover, when the null pointer is set, the registered callback function is deleted.</p>
Notes	<p>The callback function to register by this function is the different function with the callback function for swapping detection.</p> <p>The callback function registered by this function is not called at swapping detection.</p> <p>The error of this function cannot be obtained by the sd_get_error function.</p>
Usage example	<pre>/* Protocol status confirmation interrupt callback function registration example */ #include "r_sdif.h" int32_t my_sd_callback(int32_t sd_port, int32_t rsvd) { /* Getting-up processing of SD status confirmation waiting etc. */ } void func(void) { /* Registration of callback function for status confirmation*/ sd_set_intcallback(0, my_sd_callback); sd_mount(0 ,SD_MODE_POLL SD_MODE_DMA SD_MODE_DS SD_MODE_VER2X ,SD_VOLT_3_3); /* Omitted */ sd_unmount(0); /* Deletion of callback function for status confirmation */ sd_set_intcallback(0, NULL); }</pre>

sd_int_handler

Library function

Card interrupt handler

```
Format      #include "r_sdif.h"
            void sd_int_handler(int32_t sd_port)
                int32_t sd_port          | SDHI channel number (0 or 1)
```

Return values	None
---------------	------

Description	It is the card interrupt handler.
-------------	-----------------------------------

It should be embedded in the system as the processing routine of interrupt factor corresponding to the SD host controller when either one of the card swapping detection interrupt or the SD protocol status interrupt is used.

When the swapping interrupt callback function and the status confirmation interrupt callback function are registered, the callback function is called from inside of this function.

4.2.19 sd_check_int**sd_check_int**

Library function

Card interrupt request confirmation

Format `#include "r_sdif.h"`
 `int32_t sd_check_int(int32_t sd_port)`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)

Return values `SD_OK` : With the interrupt request.
 `SD_ERR` : Without the interrupt request.

Description The generation of the card interrupt request is confirmed.
 When the interrupt request from the card has been generated, the `SD_OK` is returned.
 When the interrupt request from the card has not been generated, the `SD_ERR` is returned.
 It is used when the status of the SD protocol is confirmed in the target CPU interface function `sddev_int_wait` function.

Notes This function should be used in the `sddev_int_wait` function only.

Usage example `/* Card interrupt request checking example */`
 `#include "r_sdif.h"`

 `int32_t sddev_int_wait(int32_t sd_port, int32_t time)`
 `{`
 `/* Interrupt request waiting */`
 `while(1)`
 `{`
 `if(sd_check_int(sd_port) == SD_OK)`
 `{`
 `/* There is an interrupt request */`
 `break;`
 `}`
 `/* Time-out processing etc. */`
 `}`
 `return SD_OK;`
 `}`

4.2.20 sd_get_reg**sd_get_reg**

Library function

Card register acquisition

Format	<pre>#include "r_sdif.h" int32_t sd_get_reg(int32_t sd_port, uint8_t *ocr, uint8_t *cid, uint8_t *csd, uint8_t *dsr, uint8_t *scr) int32_t sd_port I SDHI channel number (0 or 1) uint8_t *ocr O The OCR register content storage destination uint8_t *cid O The CID register content storage destination uint8_t *csd O The CSD register content storage destination uint8_t *dsr O The DSR register content storage destination uint8_t *scr O The SCR register content storage destination</pre>	
Return values	SD_OK	: Normal end
	SD_ERR	: Error
Description	<p>The contents of each card register are stored in the area shown by the argument ocr, cid, csd, dsr, and scr. Table 4.6 to Table 4.10 list the bit information in registers stored in the register value storage area.</p> <p>The area size shown in Table 4.5 is necessary for each area. The area of the storage destination should be retained in the call of this function.</p> <p>However, when the argument is a null pointer, the value is not stored.</p> <p>The data of each register is enabled only when the sd_mount function ends normally. The value of each register is disabled when the sd_mount function is not executed or when it is not normally end.</p> <p>Because the DSR register is the option register, 0 is always stored for the card which the DSR register is not mounted.</p> <p>The SCR register is exclusive for the SD memory card. 0 is always stored for the MMC card.</p>	
Notes	The error by the sd_get_error function cannot be obtained.	

Usage example

```
/* OCR register and CID register value acquisition example */
#include "r_sdif.h"

void func(void)
{
    uint8_t ocr[4], cid[16];

    /* Obtaining the register information (CSD,DSR,SCR register is not
    obtained) */
    if(sd_get_reg(0, ocr, cid, 0, 0, 0) != SD_OK)
    {
        /* Obtaining the information is failed */
    }
}
```

Table 4.5 Register Value Storage Area

Argument Name	Corresponding Register	Necessary Area Size
ocr	OCR register	4 bytes
cid	CID register	16 bytes
csd	CSD register	16 bytes
dsr	DSR register	2 bytes
scr	SCR register	8 bytes

Table 4.6 Bit Information in the OCR Register Stored in the OCR Register Value Storage Area

Byte Offset in the OCR Register Value Storage Area	Bit Information in the OCR Register to Store
0	[31:24]
1	[23:16]
2	[15:8]
3	[7:0]

Table 4.7 Bit Information in the CID Register Stored in the CID Register Value Storage Area

Byte Offset in the CID Register Value Storage Area	Bit Information in the CID Register to Store
0	ALL "0"
1	[127:120]
2	[119:112]
3	[111:104]
4	[103:96]
5	[95:88]
6	[87:80]
7	[79:72]
8	[71:64]
9	[63:56]
10	[55:48]
11	[47:40]
12	[39:32]
13	[31:24]
14	[23:16]
15	[15:8]

Table 4.8 Bit Information in the CSD Register Stored in the CSD Register Value Storage Area

Byte Offset in the CSD Register Value Storage Area	Bit Information in the CSD Register to Store
0	ALL "0"
1	[127:120]
2	[119:112]
3	[111:104]
4	[103:96]
5	[95:88]
6	[87:80]
7	[79:72]
8	[71:64]
9	[63:56]
10	[55:48]
11	[47:40]
12	[39:32]
13	[31:24]
14	[23:16]
15	[15:8]

Table 4.9 Bit Information in the DSR Register Stored in the DSR Register Value Storage Area

Byte Offset in the DSR Register Value Storage Area	Bit Information in the DSR Register to Store
0	[15:8]
1	[7:0]

Table 4.10 Bit Information in the SCR Register Stored in the SCR Register Value Storage Area

Byte Offset in the SCR Register Value Storage Area	Bit Information in the SCR Register to Store
0	[63:56]
1	[55:48]
2	[47:40]
3	[39:32]
4	[31:24]
5	[23:16]
6	[15:8]
7	[7:0]

4.2.21 sd_get_rca**sd_get_rca**

Library function

RCA register acquisition

Format	<pre>#include "r_sdif.h" int32_t sd_get_rca(int32_t sd_port, uint8_t *rca) int32_t sd_port I SDHI channel number (0 or 1) uint8_t *rca O The RCA register content storage destination</pre>		
Return values	SD_OK	:	Normal end
	SD_ERR	:	Error
Description	<p>The RCA register content of card is stored in the area indicated by the argument rca. Table 4.11 lists the bit information of the register to be stored in the RCA register value storage area.</p> <p>The area for 2-byte is necessary for RCA storage. The area of the storage destination should be retained in the call of this function.</p> <p>However, when the argument is a null pointer, the value is not stored.</p> <p>The data of RCA register is enabled only when the sd_mount function ends normally. The value of each register is disabled when the sd_mount function is not executed or when it is not normally end.</p>		
Notes	The error by the sd_get_error function cannot be obtained.		
Usage example	<pre>/* RCA register value acquisition example */ #include "r_sdif.h" void func(void) { uint8_t rca[2]; /* Obtaining information of RCA register */ if(sd_get_rca(0, rca) != SD_OK){ /* Obtaining the information is failed */ } }</pre>		

Table 4.11 Bit Information in the RCA Register Stored in the RCA Register Value Storage Area

Byte Offset in the RCA Register Value Storage Area	Bit Information in the RCA Register to Store
0	[15:8]
1	[7:0]

4.2.22 sd_get_sdstatus**sd_get_sdstatus**

Library function

SD status acquisition

Format	<pre>#include "r_sdif.h" int32_t sd_get_sdstatus(int32_t sd_port, uint8_t *sdstatus) int32_t sd_port I SDHI channel number (0 or 1) uint8_t *sdstatus O Storage destination for contents of SD status register</pre>		
Return values	SD_OK	:	Normal end
	SD_ERR	:	Error
Description	<p>Stores the contents of the 16 bytes of the card's SD status register in the area indicated by the argument sdstatus.</p> <p>An area 16 bytes in size is required for data storage. The storage destination area should be reserved by the application calling this function. However, no value is stored if the argument is a null pointer.</p> <p>The data from SD status is valid only if the sd_mount function ends normally. The value of each register is invalid if the sd_mount function was not executed or if it did not end normally.</p> <p>The SD status register is only used by SD memory cards. On MMC cards it contains all zeros.</p>		
Notes	The error by the sd_get_error function cannot be obtained.		
Usage example	<pre>/* Obtaining the contents of SD status */ #include "r_sdif.h" void func(void) { uint8_t sdstatus[16]; /* Obtaining the information of SD STATUS */ if(sd_get_sdstatus(0, sdstatus) != SD_OK) { /* Obtaining the information is failed */ } }</pre>		

4.2.23 sd_get_error**sd_get_error**

Library function

Driver error acquisition

Format

```
#include "r_sdif.h"
int32_t sd_get_error(int32_t sd_port)
    int32_t sd_port          I    SDHI channel number (0 or 1)
```

Return values

Error code

Description

When the library functions; the sd_mount function, the sd_read_sect function, and the sd_write_sect function are executed, the error code of the error generated is returned. Refer to 3.12, Error Codes for the error code. It is used when the error details of SD driver are obtained in the application program.

Usage example

```
/* Driver error acquisition example */
#include "r_sdif.h"

void func(void)
{
    int32_t err_code;

    if(sd_read_sect() < 0)
    {
        /* Obtaining the error of SD driver */
        err_code = sd_get_error(0);
        if(err_code != SD_OK)
        {
            /* SD driver error is generated */
        }
    }
}
```

4.2.24 sd_set_cdtime**sd_set_cdtime**

Library function

Card detection time setting

Format

```
#include "r_sdif.h"
int32_t sd_set_cdtime(int32_t sd_port, uint16_t cdtime)
    int32_t sd_port          | SDHI channel number (0 or 1)
    uint16_t cdtime          | Card detection time set value (0x0000 to 0x000e)
```

Return values

SD_OK	: Normal end
SD_ERR	: Error end

Description

The count value for the card detection is specified. The card detection time is determined by the following expression depending on the count value specified with cdtime.

$$\text{Card detection time} = \text{CLK} \times 2^{10+\text{cdtime}}$$

CLK: SD host controller operation clock (SD_CLK) cycle duration

The count value should be specified within the range of 0x0000 to 0x000e.

The initial value of count value is 0x000e. When the sd_init function is executed, the count value is initialized.

Usage example

```
/* Card detection time setting example */
#include "r_sdif.h"

void func(void)
{
    if(sd_set_cdtime(0, 0x0a) == SD_OK)
    {
        /* Card detection time setting is successful */
    }
    else
    {
        /* Card detection time setting is failed */
    }
}
```

4.2.25 sd_set_responsetime**sd_set_responsetime**

Library function

Response timeout time setting

Format `#include "r_sdif.h"`
 `int32_t sd_set_responsetime(int32_t sd_port, uint16_t responsetime)`
 `int32_t sd_port` | SDHI channel number (0 or 1)
 `uint16_t responsetime` | Response timeout setting value (0x0000 to 0x000f)

Return `SD_OK` : Normal end
 values `SD_ERR` : Error end

Description The response timeout time is set.

The count value for the response timeout detection is specified. The response timeout detection time is determined by the count value specified by `responsetime` with the formula below.

Response timeout detection time = $\text{SDCLK} \times 2^{13+\text{responsetime}}$
 SDCLK: SD clock

The count value should be specified in the range of 0x0000 to 0x000f.
 The initial value of count value is 0x000e. The count value is initialized when the `sd_init` function is executed.

Usage
example `/* Response timeout duration setting example */`
 `#include "r_sdif.h"`

 `void func(void)`
 `{`
 `if(sd_set_responsetime(0, 0x000a) == SD_OK)`
 `{`
 `/* Setting is successful */`
 `}`
 `else`
 `{`
 `/* Setting is failed */`
 `}`
 `}`

4.2.26 sd_get_ver**sd_get_ver**

Library function

Library version acquisition

Format	#include "r_sdif.h"		
	int32_t sd_get_ver(int32_t sd_port, uint16_t *sdhi_ver, char_t *sddrv_ver)		
	int32_t sd_port	I	SDHI channel number (0 or 1)
	uint16_t *sdhi_ver	O	Pointer that indicates SD host controller IP version storage destination
	char_t *sddrv_ver	O	Pointer that indicates SD driver library version storage destination

Return values	SD_OK	: Normal end
	SD_ERR	: Error end

Description The version register content of SD host controller IP is stored in the area indicated by sdhi_ver.

The version of this library is stored in the area indicated by sddrv_ver in ASCII character string. The area for 32 bytes is necessary as the storage area of the SD driver library version.

When the pointer that indicates the storage destination is NULL, that information is not stored.

Usage example

```

/* Library version acquisition example */
#include "r_sdif.h"

void func(void)
{
    uint16_t ip_ver;
    char_t lib_ver[32];

    /* Version obtaining */
    sd_get_ver(0, &ip_ver, lib_ver);

    printf("The version of IP is 0x%04x\n", ip_ver);
    printf("The version of library is %s\n", lib_ver);
}

```

4.2.27 sd_lock_unlock**sd_lock_unlock**

Library function

Card locking/unlocking

Format `#include "r_sdif.h"``int32_t sd_lock_unlock(int32_t sd_port, uint8_t code, uint8_t *pwd, uint8_t len)`

<code>int32_t sd_port</code>		SDHI channel number (0 or 1)
<code>uint8_t code</code>		Operation code
<code>uint8_t *pwd</code>		Password
<code>uint8_t len</code>		Length of password (by bytes) (1 to 16 bytes)

Return values `SD_OK` : Normal end`SD_ERR` : Error end

Description Locks or unlocks the card using the operation code specified by the argument code. The operation codes are listed in Table 4.12 and Table 4.13.

Notes To prevent a situation in which the card cannot be unlocked due to a password entry error, it is not possible to set the password (bit 0: SET_PWD = 1) when the card is in the locked state. When this occurs, the function terminates with an error. Always unlock the card before updating the password.

It is not possible to update the password if the total length of the old and new passwords exceeds 16 bytes. In such cases, first clear the password (bit 1: CLR_PWD = 1), then set a new password (bit 0: SET_PWD = 1).

Usage example

```
/* Card locking/unlocking example */
#include <stdio.h>
#include "r_sdif.h"

void func(void)
{
    uint8_t code;
    uint8_t pwd[2];

    /* Sets the password to "12" */
    pwd[0] = 0x31;
    pwd[1] = 0x32;
    /* Locks the card with the password "12" */
    code = 0x05;
    if(sd_lock_unlock(0, code, pwd, sizeof(pwd)) != SD_OK)
    {
        printf("Locking failed\n");
        return;
    }
    printf("The card is locked\n");
}
```

Table 4.12 Operation Codes (1)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Set to 0				ERASE	LOCK_ UNLOCK	CLR_PWD	SET_PWD

Table 4.13 Operation Codes (2)

Item	Description	Remarks
ERASE	1: Forced Erase Operation	Other bits (Bits 2 to 0) are ignored
LOCK_UNLOCK	1: Performs Lock Operation 0: Performs Unlock Operation	Enables to execute simultaneously with Set New password to PWD operation
CLR_PWD	1: Performs Clears PWD operation	-
SET_PWD	1: Performs Set New password to PWD operation	-

Note: For the details on each operation, refer to SD PHYSICAL LAYER SPECIFICATION.

4.2.28 sd_get_speed**sd_get_speed**

Library function

Card speed acquisition

Format `#include "r_sdif.h"`
 `int32_t sd_get_speed(int32_t sd_port, uint8_t *cls, uint8_t *move)`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)
 `uint8_t *cls` **O** Pointer indicating storage area of speed class
 `uint8_t *move` **O** Pointer indicating storage area of transfer speed

Return `SD_OK` : Normal end
 values `SD_ERR` : Error end

Description Stores the speed class, and transfer speed of the mounted card in the areas specified by `cls`, and `move`, respectively.

The speed class value of the mounted card is stored in `cls`.

Table 4.14 Speed Classes

Speed Class	Value
Class 0 (C0)	0x00
Class 2 (C2)	0x01
Class 4 (C4)	0x02
Class 6 (C6)	0x03

The minimum speed [MB/sec.] of the mounted card when the library is in Default-Speed mode is stored in `move`. (A value is stored for speed classes 2 to 6 only.)

**Usage
example**

```
/* Speed class acquisition example */
#include <stdio.h>
#include "r_sdif.h"

void func(void)
{
    uint8_t  class;

    /* Acquires the speed class and UHS speed class */
    sd_get_speed(0, &class, NULL);

    /* Determines the speed class */
    if(class == 0x01)
    {
        printf("The speed class is 2\n");
    }
    else if(class == 0x02)
    {
        printf("The speed class is 4\n");
    }
    else if(class == 0x03)
    {
        printf("The speed class is 6\n");
    }
    else
    {
        printf("The speed class is 0\n");
    }
}
```

4.2.29 sdio_get_cia**sdio_get_cia**

Library function

Obtaining SDIO card CIA information

Format	<pre>#include "r_sdif.h" int32_t sdio_get_cia(int32_t sd_port, uint8_t *reg, uint8_t *cis, uint32_t func_num, int32_t cnt);</pre>	
	int32_t sd_port	I SDHI channel number (0 or 1)
	uint8_t *reg	O CCCR (function0) or FBR (except function 0) register content storage destination (area for 32 bytes)
	uint8_t *cis	O CIS content storage destination
	uint32_t func_num	I Function number
	int32_t cnt	I Byte count of CIS read
Return values	SD_OK	: Normal end
	Excluding SD_OK	: Error end (Refer to 3.12, Error Codes for details)
Description	<p>Stores the content of the CCCR register (Card Common Control Registers) and the FBR register (Function Basic Registers) with the function number of the SDIO card specified by the argument func_num in the area specified by the argument reg. The area for 32 bytes should be retained as the area specified by the argument reg. This function also stores the CIS in the area specified by the argument cis. The area for the bytes specified by the argument cnt should be retained as the area specified by the argument cis.</p> <p>Note that no value can be stored if the argument is a null pointer.</p>	
Notes	-	

4.2.30 sdio_get_ioocr

sdio_get_ioocr

Library function

Obtaining SDIO card IO_OCR information

Format	<pre>#include "r_sdif.h" int32_t sdio_get_ioocr(int32_t sd_port, uint32_t *ioocr); int32_t sd_port I SDHI channel number (0 or 1) uint32_t * ioocr O R4 Response content storage destination, Refer to Table 4.15</pre>		
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	<p>Stores the content of the Response (R4) obtained by issuing IO_SEND_OP_CNOD command to the SDIO card in the area specified by the argument ioocr when the SDIO card is mounted. Note that no value can be stored if the argument is a null pointer. The data of the Response (R4) is valid only when the sd_mount function normally ends. The values in the registers are invalid if the sd_mount function is not executed or ends with error.</p>		
Notes	The error by the sd_get_error function cannot be obtained.		

Table 4.15 ioocr content to be stored

ioocr[31]	ioocr[30]	ioocr[29]	ioocr[28]	ioocr[27]	ioocr[26]	ioocr[25]	ioocr[24]
C	Number of I/O Functions			Memory Present	-	-	S18A
ioocr[23]	ioocr[22]	ioocr[21]	ioocr[20]	ioocr[19]	ioocr[18]	ioocr[17]	ioocr[16]
I/O OCR[23:16]							
ioocr[15]	ioocr[14]	ioocr[13]	ioocr[12]	ioocr[11]	ioocr[10]	ioocr[9]	ioocr[8]
I/O OCR[15:8]							
ioocr[7]	ioocr[6]	ioocr[5]	ioocr[4]	ioocr[3]	ioocr[2]	ioocr[1]	ioocr[0]
I/O OCR[7:0]							

4.2.31 sdio_get_ioinfo**sdio_get_ioinfo**

Library function

Obtaining SDIO card IO information

Format	<pre>#include "r_sdif.h" int32_t sdio_get_ioinfo(int32_t sd_port, uint8_t *ioinfo); int32_t sd_port I SDHI channel number (0 or 1) uint32_t * ioinfo O R4 Response content [31:24] storage destination, Refer to Table 4.16</pre>		
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	Stores the bits from 31 to 24 in the Response (R4) content obtained by issuing IO_SEND_OP_CNOD command to the SDIO card in the area specified by the argument ioinfo when the SDIO card is mounted. Note that no value can be stored if the argument is a null pointer. The data of the Response (R4) is valid only when the sd_mount function normally ends. The values in the registers are invalid if the sd_mount function is not executed or ends with error.		
Notes	The error by the sd_get_error function cannot be obtained.		

Table 4.16 ioinfo content to be stored

ioinfo[7]	ioinfo[6]	ioinfo[5]	ioinfo[4]	ioinfo[3]	ioinfo[2]	ioinfo[1]	ioinfo[0]
C	Number of I/O Functions			Memory Present	-	-	S18A

4.2.32 sdio_reset

sdio_reset

Library function

SDIO reset

Format	<pre>#include "r_sdif.h" int32_t sdio_reset(int32_t sd_port); int32_t sd_port I SDHI channel number (0 or 1)</pre>		
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	Writes 1 to the bit 3 (RES) in the CCCR register 06h (I/O Abort) of the SDIO card connected to the SDHI channel specified by the argument sd_port and executes the SDIO reset.		
Notes	When the SDIO reset is executed by this function, mount the card by using the sd_mount function before the card access processing later required.		

4.2.33 sdio_set_enable**sdio_set_enable**

Library function

SDIO function enable

Format `#include "r_sdif.h"`
 `int32_t sdio_set_enable(int32_t sd_port, uint8_t func_bit);`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)
 `uint8_t func_bit` **I** I/O enabled (Bit map), Refer to Table 4.17

Return `SD_OK` : Normal end
 values Excluding `SD_OK` : Error end (Refer to 3.12, Error Codes for details)

Description Writes the value specified by the argument `func_bit` to the CCCR register 02h (I/O Enable) of the SDIO card connected to the SDHI channel specified by the argument `sd_port`.

Notes

Table 4.17 func_bit Setting

func_bit[7]	func_bit[6]	func_bit[5]	func_bit[4]	func_bit[3]	func_bit[2]	func_bit[1]	func_bit[0]
IOE7	IOE6	IOE5	IOE4	IOE3	IOE2	IOE1	–

4.2.34 sdio_get_ready**sdio_get_ready**

Library function

SDIO ready confirmation

Format `#include "r_sdif.h"`
 `int32_t sdio_get_ready(int32_t sd_port, uint8_t *func_bit);`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)
 `uint8_t *func_bit` **O** I/O Ready (bit map), Refer to Table 4.18

Return `SD_OK` : Normal end
 values Excluding `SD_OK` : Error end (Refer to 3.12, Error Codes for details)

Description Stores the value of the CCCR register 03h (I/O Ready) of the SDIO card connected to the SDHI channel specified by the argument `sd_port` to the area specified by the argument `func_bit`.

Notes

Table 4.18 func_bit Setting

func_bit[7]	func_bit[6]	func_bit[5]	func_bit[4]	func_bit[3]	func_bit[2]	func_bit[1]	func_bit[0]
IOR7	IOR6	IOR5	IOR4	IOR3	IOR2	IOR1	–

4.2.35 sdio_set_blocklen**sdio_set_blocklen**

Library function

SDIO block size setting

Format	<pre>#include "r_sdif.h" int32_t sdio_set_blocklen(int32_t sd_port, uint16_t len, uint32_t func_num); int32_t sd_port SDHI channel number (0 or 1) uint16_t len SDIO block size (32, 64, 128, 256, and 512 bytes) uint32_t func_num Function number</pre>		
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	<p>Sets the SDIO block size of the function specified by the argument func_num to the SDIO card connected to the SDHI channel specified by the argument sd_port.</p> <p>When the argument func_num is 0, the SDIO block size specified by the argument len is written to the CCCR register 10-11h (FN0 Block Size).</p> <p>In other cases, the SDIO block size is written to the FBR register offset (10-11h: I/O Block Size).</p> <p>When the value except 64, 128, 256, or 512 bytes is specified for the argument len, SD_ERR is set to the return value without any processing.</p>		
Notes	-		

4.2.36 sdio_get_blocklen**sdio_get_blocklen**

Library function

Obtaining SDIO block size

Format	<pre>#include "r_sdif.h" int32_t sdio_get_blocklen(int32_t sd_port, uint16_t *len, uint32_t func_num); int32_t sd_port I SDHI channel number (0 or 1) uint16_t *len O SDIO block size uint32_t func_num I Function number</pre>	
Return values	SD_OK	: Normal end
	Excluding SD_OK	: Error end (Refer to 3.12, Error Codes for details)
Description	Obtains the SDIO block size of the function specified by the argument func_num in the SDIO card connected to the SDHI channel specified by the argument sd_port, and stores it in the area specified by the argument len. When the argument func_num is 0, the value set to the CCCR register 10-11h (FN0 Block Size) is stored in the area specified by the argument len. In other cases, the value set to the FBR register offset (10-11h: I/O Block Size) is stored in the said area.	
Notes	-	

4.2.37 sdio_set_int**sdio_set_int**

Library function

SDIO interrupt setting

Format	<pre>#include "r_sdif.h" int32_t sdio_set_int(int32_t sd_port, uint8_t func_bit, int32_t enab); int32_t sd_port SDHI channel number (0 or 1) uint8_t func_bit SDIO interrupt setting (bit map), Refer to Table 4.19 int32_t enab 0: SDIO Interrupts disabled 1: SDIO Interrupts enabled</pre>		
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	<p>Writes the value specified by the argument <code>func_bit</code> to the bits from 7 to 1 in the CCCR register 04h (Int Enable) of the SDIO card connected to the SDHI channel specified by the argument <code>sd_port</code>. At this point, 1 is set to the bit 0 (IENM) of the CCCR register 04h if the argument <code>enab</code> specifies 1, or 0 is set if the said argument specifies 0.</p> <p>The SDIO interrupt enable (SDHI module) function (<code>sdio_enable_int</code>) is executed in this function.</p>		
Notes	-		

Table 4.19 func_bit and enab Setting

func_bit[7]	func_bit[6]	func_bit[5]	func_bit[4]	func_bit[3]	func_bit[2]	func_bit[1]	enb
IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IENM

4.2.38 sdio_get_int**sdio_get_int**

Library function

Obtaining SDIO interrupt setting status

Format `#include "r_sdif.h"`
 `int32_t sdio_get_int(int32_t sd_port, uint8_t *func_bit, int32_t *enab);`
 `int32_t sd_port` | SDHI channel number (0 or 1)
 `uint8_t *func_bit` | SDIO Interrupts setting status (bit map), Refer to Table 4.20
 `int32_t *enab` | SDIO Interrupts setting status
 0: SDIO Interrupts disabled
 1: SDIO Interrupts enabled

Return `SD_OK` : Normal end
 values Excluding `SD_OK` : Error end (Refer to 3.12, Error Codes for details)

Description Stores the content of the CCCR register 04h (Int Enable) of the SDIO card connected to the SDHI channel specified by the argument `sd_port` in the area specified by the argument `func_bit`. At this time, 1 is set to the area specified by the argument `enab` if the bit 0 (IENM) in the CCCR register 04h specifies 1, or 0 is set if the said bit specifies 0.

Notes -

Table 4.20 func_bit and enab Setting

<code>func_bit[7]</code>	<code>func_bit[6]</code>	<code>func_bit[5]</code>	<code>func_bit[4]</code>	<code>func_bit[3]</code>	<code>func_bit[2]</code>	<code>func_bit[1]</code>	<code>func_bit[0]/ enab</code>
IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IENM

4.2.39 sdio_read_direct

sdio_read_direct

Library function

Direct-read from SDIO card

Format	#include "r_sdif.h" int32_t sdio_read_direct(int32_t sd_port, uint8_t *buff, uint32_t func, uint32_t adr); <table><tr><td>int32_t sd_port</td><td>I</td><td>SDHI channel number (0 or 1)</td></tr><tr><td>uint8_t *buff</td><td>O</td><td>Storage area for read data</td></tr><tr><td>uint32_t func</td><td>I</td><td>Function number</td></tr><tr><td>uint32_t adr</td><td>I</td><td>Read address</td></tr></table>			int32_t sd_port	I	SDHI channel number (0 or 1)	uint8_t *buff	O	Storage area for read data	uint32_t func	I	Function number	uint32_t adr	I	Read address
int32_t sd_port	I	SDHI channel number (0 or 1)													
uint8_t *buff	O	Storage area for read data													
uint32_t func	I	Function number													
uint32_t adr	I	Read address													
Return values	SD_OK	:	Normal end												
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)												
Description	Reads the data from the address specified by the argument adr of the function specified by the argument func in the SDIO card connected to the SDHI channel specified by the argument sd_port. IO_RW_DIRECT (CMD52) is used to read the data.														
Notes	-														

4.2.40 sdio_write_direct**sdio_write_direct**

Library function

Direct-write to SDIO card

Format	<pre>#include "r_sdif.h" int32_t sdio_write_direct(int32_t sd_port, uint8_t *buff, uint32_t func, uint32_t adr, uint32_t raw_flag);</pre>		
	int32_t sd_port		SDHI channel number (0 or 1)
	uint8_t *buff		Write data storage area
	uint32_t func		Function number
	uint32_t adr		Write address
	uint32_t raw_flag		RAW Flag
			SD_IO_SIMPLE_WRITE : RAW Flag=0
			SD_IO_VERIFY_WRITE : RAW Flag=1
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	<p>Writes the data to the address specified by the argument adr of the function specified by the argument func in the SDIO card connected to the SDHI channel specified by the sd_port. IO_RW_DIRECT (CMD52) is used to write the data, and RAW Flag is specified by the argument raw_flag at the time of execution. When SD_IO_VERIFY_WRITE is specified for the argument raw_flag, the value of Read or Write Data in Response (5) obtained after writing the data should be set to the write data storage area specified by the argument buff. When SD_IO_SIMPLE_WRITE is specified, the value in the write data storage area specified by the said argument after writing the data is undefined.</p>		
Notes	<p>The value of the write data storage area specified by the argument buff can be changed by executing this function.</p>		

4.2.41 sdio_read

sdio_read

Library function

Reading data from SDIO card

Format	<pre>#include "r_sdif.h" int32_t sdio_read(int32_t sd_port, uint8_t *buff, uint32_t func, uint32_t adr, int32_t cnt, uint32_t op_code);</pre>		
	int32_t sd_port	I	SDHI channel number (0 or 1)
	uint8_t *buff	O	Storage area for read data
	uint32_t func	I	Function number
	uint32_t adr	I	Read address
	int32_t cnt	I	Byte count to be read
	uint32_t op_code	I	Bit 0: OP Code SD_IO_FIXED_ADDR: OP Code=0 (address fixed) SD_IO_INCREMENT_ADDR: OP Code=1 (address incremented) Bit 4: Setting for forced byte reading from SDIO card SD_IO_FORCE_BYTE: Forced byte reading from SDIO card
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	<p>Reads the data for the byte count specified by the argument cnt from the address specified by the argument adr of the function specified by the argument func in the SDIO card connected to the SDHI channel specified by the argument sd_port. IO_RW_EXTENDED (CMD53) is used to read the data, and OP Code is specified by bit 0 of the argument op_code at the time of execution.</p> <p>When SD_IO_FORCE_BYTE is specified for bit 4 of the argument op_code, 0 is set to Block Mode during IO_RW_EXTENDED (CMD53) execution to read the bytes from the SDIO card. When SD_IO_FORCE_BYTE is not specified, 1 is set to Block Mode to read the blocks from the SDIO card for the size set by the function specified by the argument func in the card connected to the SDHI channel specified by the argument sd_port. However, if the byte count to be read specified by the argument cnt is smaller than the block size, 0 is set to Block Mode during IO_RW_EXTENDED (CMD53) execution to read the bytes from the SDIO card.</p> <p>If the SDIO block transfer is not supported or the SDIO block size is not set in the SDIO card connected to the SDHI channel specified by the argument sd_port, SD_ERR_ILL_FUNC should be set to the return value.</p>		
Notes	-		

4.2.42 sdio_write

sdio_write

Library function

Writing data to SDIO card

Format	<pre>#include "r_sdif.h" int32_t sdio_write(int32_t sd_port, uint8_t *buff, uint32_t func, uint32_t adr, int32_t cnt, uint32_t op_code);</pre>		
	int32_t sd_port		SDHI channel number (0 or 1)
	uint8_t *buff		Storage area for write data
	uint32_t func		Function number
	uint32_t adr		Write address
	int32_t cnt		Byte count to be written
	uint32_t op_code		Bit 0: OP Code SD_IO_FIXED_ADDR: OP Code=0 (address fixed) SD_IO_INCREMENT_ADDR: OP Code=1 (address incremented) Bit 4: Setting for forced byte writing to SDIO card SD_IO_FORCE_BYTE: Forced byte writing to SDIO card
Return values	SD_OK	:	Normal end
	Excluding SD_OK	:	Error end (Refer to 3.12, Error Codes for details)
Description	<p>Writes the data for the byte count specified by the argument cnt to the address specified by the argument adr of the function specified by the argument func in the SDIO card connected to the SDHI channel specified by the argument sd_port. IO_RW_EXTENDED (CMD53) is used to write the data, and OP Code is specified by bit 0 of the argument op_code at the time of execution.</p> <p>When SD_IO_FORCE_BYTE is specified for bit 4 of the argument op_code, 0 is set to Block Mode during IO_RW_EXTENDED (CMD53) execution to write the bytes to the SDIO card.</p> <p>When SD_IO_FORCE_BYTE is not specified, 1 is set to Block mode to write blocks to the SDIO card for the size set by the function specified by the argument func in the card connected to the SDHI channel specified by the argument sd_port. However, if the byte count to be written specified by the argument cnt is smaller than the block size, 0 is set to the Block Mode during IO_RW_EXTENDED (CMD53) execution to write the bytes to the SDIO card.</p> <p>If the SDIO block transfer is not supported or the SDIO block size is not set in the SDIO card connected to the SDHI channel specified by the argument sd_port, SD_ERR_ILL_FUNC should be set to the return value.</p>		
Notes	-		

4.2.43 sdio_enable_int

sdio_enable_int

Library function

SDIO Interrupt enable (SDHI module)

Format `#include "r_sdif.h"`
 `int32_t sdio_enable_int(int32_t sd_port);`
 `int32_t sd_port` `I` SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description Enables the SDIO interrupts in the SDHI channel specified by the argument `sd_port`.
 This function does not access to the SDIO card.

Notes The error by the `sd_get_error` function cannot be obtained.

4.2.44 sdio_disable_int

sdio_disable_int

Library function

SDIO Interrupt disable (SDHI module)

Format `#include "r_sdif.h"`
 `int32_t sdio_disable_int(int32_t sd_port);`
 `int32_t sd_port` `I` SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description Disables the SDIO interrupts in the SDHI channel specified by the argument `sd_port`.
 This function does not access to the SDIO card.

Notes The error by the `sd_get_error` function cannot be obtained.

4.2.45 sdio_set_intcallback**sdio_set_intcallback**

Library function

Registration of SDIO interrupt callback function

Format `#include "r_sdif.h"`
 `int32_t sdio_set_intcallback(int32_t sd_port, int32_t (*callback)(int32_t));`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)
 `int32_t` **I** Callback function to be registered
 `(*callback)(int32_t)`

Return `SD_OK` : Normal end
 values `SD_ERR` : Error end

Description Registers the SDIO interrupt callback function.
 The callback function registered by this function is called when the SDIO interrupt is generated.
 When using the callback function, it should be registered before executing the `sd_mount`
 function. When the
 callback function is not defined, it is not called in the interrupt processing.
 If the null pointer is set, the registered callback function is deleted.

Notes The error by the `sd_get_error` function cannot be obtained.

4.2.46 sdio_int_handler**sdio_int_handler**

Library function

SDIO interrupt handler

Format `#include "r_sdif.h"`
 `void sdio_int_handler(int32_t sd_port)`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)

Return `None`
 values

Description This is the SDIO interrupt handler.
 When interrupts are used for the detection method of the SDIO interrupts, this function should
 be mounted in the system as a processing routine for the interrupt source corresponding to the
 SD host controller.
 If the interrupt callback function is registered, the callback function is called by this function.

4.2.47 sdio_check_int

sdio_check_int

Library function

SDIO interrupt request confirmation

Format	<pre>#include "r_sdif.h" int32_t sdio_check_int(int32_t sd_port) int32_t sd_port I SDHI channel number (0 or 1)</pre>		
Return values	SD_OK	:	SDIO Interrupt request occurred
	SD_ERR	:	SDIO Interrupt request not occurred
Description	Confirms the generation of the SDIO interrupt request. SD_OK is returned when the SDIO interrupt request from the SDIO has been generated SD_ERR is returned when the SDIO interrupt request from the SDIO has not been generated.		
Notes	The error by the sd_get_error function cannot be obtained.		

4.2.48 sdio_abort

sdio_abort

Library function

SDIO driver termination

Format	<pre>#include "r_sdif.h" void sdio_abort(int32_t sd_port, uint32_t func_num); int32_t sd_port SDHI channel number (0 or 1) uint32_t func_num Function number</pre>
Return values	None
Description	<p>Terminates the SDIO card transfer processing.</p> <p>This function is used when the processing is suspended by the application, or the uninsertion is detected by the interrupt.</p> <p>The suspend state by this function continues until the sdio_read function and the sdio_write function are executed. When the said functions are called after executing this function, the transfer is terminated in the middle of process.</p>
Notes	<p>The content of the card is not guaranteed when the transfer is terminated during the write processing.</p>

4.2.49 sdio_set_blkcnt**sdio_set_blkcnt**

Library function

Continuous transfer block count setting

Format	<pre>#include "r_sdif.h" int32_t sdio_set_blkcnt(int32_t sd_port, int16_t blocks); int32_t sd_port SDHI channel number (0 or 1) int16_t blocks Number of continuous transferring blocks (1 to 0x7fff)</pre>		
Return values	SD_OK	:	Normal end
	SD_ERR	:	Error end
Description	<p>Sets the maximum value for the number of continuous transferring blocks. In the <code>sdio_write</code> function or the <code>sdio_read</code> function, the transfer is executed based on the number of continuous transferring blocks specified by this function. When the transfer byte count specified for the <code>sdio_write</code> function or the <code>sdio_read</code> function is smaller than the SDIO block size, the SDIO byte transfer is executed with the specified data transfer size. When the data transfer size specified for the <code>sdio_write</code> function or the <code>sdio_read</code> function is larger than the SDIO block size, the SDIO block transfer is executed with the specified SDIO block size. Furthermore, when the data transfer size is larger than the size of the SDIO block multiplied by the number of continuous transferring blocks, the SDIO block transfer is executed by dividing the data into each number of continuous transferring blocks. If the SDIO data smaller than the SDIO block size is left, the SDIO byte transfer is continuously executed. The initial value for the number of continuous transferring blocks is 32. If the number of continuous transferring blocks is not set by this function, the initial value is set when the <code>sd_init</code> function is executed. The number of continuous transferring blocks can be specified for the argument blocks is from 1 to 0x7fff. When the number of blocks does not fall within this range, <code>SD_ERR</code> is returned.</p>		
Notes	<p>It is necessary to initialize this function by the <code>sd_init</code> function before it is executed. The error by the <code>sd_get_error</code> function cannot be obtained.</p>		

4.2.50 sdio_get_blkcnt

sdio_get_blkcnt

Library function

Continuous transfer block count acquisition

Format	<pre>#include "r_sdif.h" int32_t sdio_get_blkcnt(int32_t sd_port); int32_t sd_port SDHI channel number (0 or 1)</pre>		
Return values	≥ 1	:	Number of continuous transferring blocks
	SD_ERR	:	Error end
Description	Returns the number of continuous transferring blocks for the SDIO card. SD_ERR is returned when this function is executed before initialization by the sd_init function.		
Notes	It is necessary to initialize this function by the sd_init function before it is executed. The error by the sd_get_error function cannot be obtained.		

4.3 Target CPU Interface Functions

To embed the SD driver in the system, the target CPU interface function corresponding to the target system must be made. The target CPU interface function list is shown in Table 4.21.

Table 4.21 Target CPU Interface Functions

Function Name	Function Outline
sddev_init	Initialization of hardware
sddev_finalize	Termination of hardware
sddev_power_on	Starting of power supply to card
sddev_power_off	Stopping of power supply to card
sddev_read_data	Data read processing
sddev_write_data	Data write processing
sddev_get_clockdiv	Clock frequency dividing ratio acquisition
sddev_set_port	Port setting for card
sddev_int_wait	Card interrupt standby
sddev_loc_cpu	Card interrupt disable
sddev_unl_cpu	Card interrupt enable
sddev_init_dma	Data transfer DMA initialization
sddev_wait_dma_end	Data transfer DMA transfer completion standby
sddev_disable_dma	Data transfer DMA disable
sddev_reset_dma	Reset of DMA
sddev_finalize_dma	Termination of DMA
sddev_cmd0_sdio_mount	Selecting CMD0 issuance when SDIO card is mounted
sddev_cmd8_sdio_mount	Selecting CMD8 issuance when SDIO card is mounted

4.3.1 sddev_init

sddev_init

Target CPU Interface Function

Initialization of hardware

Format `#include "r_sdif.h"`
 `int32_t sddev_init(int32_t sd_port)`
 `int32_t sd_port` `I` SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description Initializes the hardware resource of CPU required for the card access besides the SD host controller. This function is called by the library function `sd_init` function.
The pins for card insertion detection (CD pin) should be enabled, and the SD host controller interrupt should be set if necessary.

Notes 2²⁴ clocks (a standard clock is a supply clock to the SD host controller) is necessary as the time from the start of the card insertion detection to the recognition of card insertion (hereinafter called "card detection time"). The insertion detection begins at the time that the pins for the card insertion detection of the SD host controller are enabled. Therefore, the other library functions should be executed after the card detection time has passed since the `sd_init` function was executed. When the other library functions are executed before the card detection time is passed, "card uninsertion error" might occur regardless of insertion/uninsertion of the card.
The card detection time can be changed by the library function `sd_set_cdtime` function.

The `SD_MODE_HWINT` should be specified for the status confirmation method specified by the `sd_mount` function when the SD host controller interrupt is enabled. In this case, the software polling cannot be used.

4.3.2 sddev_finalize

sddev_finalize		Target CPU Interface Function
Termination of hardware		

Format	<pre>#include "r_sdif.h" int32_t sddev_finalize(int32_t sd_port); int32_t sd_port I SDHI channel number (0 or 1)</pre>	
Return values	SD_OK	: Normal end
Description	Terminates the hardware related to the card. This function is called by the library function sd_finalize function. If the termination process is required in the peripheral I/O set by the sddev_init function, it is executed in this function.	

4.3.3 sddev_power_on

sddev_power_on

Target CPU Interface Function

Starting of power supply to card

Format `#include "r_sdif.h"`
 `int32_t sddev_power_on(int32_t sd_port)`
 `int32_t sd_port` `I` SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description Supplies power to the card. This function is called by the `sd_mount` function.
 The waiting time should be retained till the power voltage reaches the operable level after
 the power supply to retain the initialization time for the card.
 Refer to the SD Memory Card Specifications Part 1 PHYSICAL LAYER SPECIFICATION for
 the details of power-up sequence.

4.3.4 sddev_power_off

sddev_power_off

Target CPU Interface Function

Stopping of power supply to card

Format `#include "r_sdif.h"`
 `int32_t sddev_power_off(int32_t sd_port)`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description Stops the power supply to the SD card.
 Refer to the SD Memory Card Specifications Part 1 PHYSICAL LAYER SPECIFICATION for
 the details of power down sequence.

4.3.5 sddev_read_data**sddev_read_data**

Target CPU Interface Function

Data read processing

Format	<pre>#include "r_sdif.h" int32_t sddev_read_data(int32_t sd_port, uint8_t *buff, uint32_t reg_addr, int32_t num);</pre> <div> <div>int32_t sd_port</div> <div>I</div> <div>SDHI channel number (0 or 1)</div> </div> <div> <div>uint8_t *buff</div> <div>O</div> <div>Pointer that indicates the read data storage destination</div> </div> <div> <div>uint32_t reg_addr</div> <div>I</div> <div>Host controller I/P data register address</div> </div> <div> <div>int32_t num</div> <div>I</div> <div>Read-out byte count</div> </div>		
Return values	SD_OK	:	Normal end
	SD_ERR	:	Error end
Description	<p>Reads out the sector data from the data register reg_addr for the byte count for num and stores the data in the area shown by buff.</p> <p>The address of 8-byte alignment is specified as the data register address. The data register address should be converted to the uint64_t type pointer to access.</p> <p>The maximum value of the read-out byte count is 512 bytes.</p>		
Notes	<p>Byte alignment at the read-out data storage destination depends on the application program. When the DMA transfer is selected, this function must be mounted.</p>		
Creation example	<pre>/* Data read processing example */ #include "r_sdif.h" int32_t sddev_read_data(int32_t sd_port ,uint8_t *buff ,uint32_t reg_addr ,int32_t num) { int32_t i; int32_t cnt; uint64_t *reg; uint64_t *ptr_l; uint8_t *ptr_c; volatile uint64_t tmp; reg = (uint64_t *)(reg_addr); cnt = (num / 8); if(((uint32_t)buff & 0x7uL) != 0uL) { ptr_c = (uint8_t *)buff; for(i = cnt; i > 0 ; i--) { tmp = *reg; *ptr_c++ = (uint8_t)(tmp); *ptr_c++ = (uint8_t)(tmp >> 8); *ptr_c++ = (uint8_t)(tmp >> 16); *ptr_c++ = (uint8_t)(tmp >> 24); *ptr_c++ = (uint8_t)(tmp >> 32); *ptr_c++ = (uint8_t)(tmp >> 40); *ptr_c++ = (uint8_t)(tmp >> 48); *ptr_c++ = (uint8_t)(tmp >> 56); } } }</pre>		

```
        cnt = (num % 8);
        if(cnt != 0)
        {
            tmp = *reg;
            for(i = cnt; i > 0 ; i--)
            {
                *ptr_c++ = (uint8_t)(tmp);
                tmp >>= 8;
            }
        }
    }
    else
    {
        ptr_l = (uint64_t *)buff;
        for(i = cnt; i > 0 ; i--)
        {
            *ptr_l++ = *reg;
        }

        cnt = (num % 8);
        if(cnt != 0)
        {
            ptr_c = (uint8_t *)ptr_l;
            tmp = *reg;
            for(i = cnt; i > 0 ; i--)
            {
                *ptr_c++ = (uint8_t)(tmp);
                tmp >>= 8;
            }
        }
    }

    return SD_OK;
}
```

4.3.6 sddev_write_data

sddev_write_data

Target CPU Interface Function

Data write processing

Format	<pre>#include "r_sdif.h" int32_t sddev_write_data(int32_t sd_port, uint8_t *buff, uint32_t reg_addr, int32_t num); int32_t sd_port SDHI channel number (0 or 1) uint8_t *buff Pointer that indicates the writing data storage destination uint32_t reg_addr Host controller I/P data register address int32_t num Write-in byte count</pre>		
Return values	SD_OK	:	Normal end
	SD_ERR	:	Error end
Description	<p>Writes the sector data shown by buff to the data register reg_addr for num byte. The address of 8-byte alignment is specified as the data register address. The data register address should be converted to the uint64_t type pointer to access. The maximum value of the write-in byte count is 512 bytes.</p>		
Notes	<p>The data register must be accessed by the length of 64-bit. Byte alignment at the writing data storage destination depends on the application program. When the DMA transfer is selected, this function must be mounted.</p>		

Creation
example

```
/* Data write processing example */
#include "r_sdif.h"

int32_t sddev_write_data(int32_t sd_port, uint8_t *buff, uint32_t reg_addr
                        ,int32_t num)
{
    int32_t i;
    uint64_t *reg = (uint64_t *) (reg_addr);
    uint64_t *ptr = (uint64_t *) buff;
    uint64_t tmp;

    num += 7;
    num /= 8;
    if(((uint32_t)buff & 0x7) != 0)
    {
        for(i = num; i > 0 ; i--)
        {
            tmp = *buff++ ;
            tmp |= *buff++ << 8;
            tmp |= *buff++ << 16;
            tmp |= *buff++ << 24;
            tmp |= *buff++ << 32;
            tmp |= *buff++ << 40;
            tmp |= *buff++ << 48;
            tmp |= *buff++ << 56;
            *reg = tmp;
        }
    }
    else
    {
        for(i = num; i > 0 ; i--)
        {
            *reg = *ptr++;
        }
    }
    return SD_OK;
}
```

4.3.7 sddev_get_clockdiv**sddev_get_clockdiv**

Target CPU Interface Function

Clock frequency dividing ratio acquisition

Format `#include "r_sdif.h"`
 `uint32_t sddev_get_clockdiv(int32_t sd_port, int32_t clock);`
 `int32_t sd_port` | SDHI channel number (0 or 1)
 `int32_t clock` | Set clock frequency
 SD_CLK_400KHZ
 SD_CLK_1MHZ
 SD_CLK_5MHZ
 SD_CLK_10MHZ
 SD_CLK_20MHZ
 SD_CLK_25MHZ
 SD_CLK_50MHZ

Return Clock frequency dividing ratio
 values

Description The frequency dividing ratio of the clock (SDCLK) supplied to the card is determined and the value is returned.
 The upper limit value of the clock frequency that should be set is set to the argument clock.
 The clock frequency dividing ratio should be determined to become a value close to the clock frequency clock to be set by the operation clock to the SD host controller.

The clock frequency dividing ratio and SDCLK are shown in Table 4.22.
 The macro definition of the clock frequency dividing ratio should be taken as the return value.

Table 4.22 Clock Dividing Ratio

Frequency Dividing Ratio	Macro Definition of Clock Frequency Dividing Ratio	SDCLK at 132 MHz Operation Clock
4	SD_DIV_4	33 MHz
8	SD_DIV_8	16.5 MHz
16	SD_DIV_16	8.3 MHz
32	SD_DIV_32	4.1 MHz
64	SD_DIV_64	2.1 MHz
128	SD_DIV_128	1.0 MHz
256	SD_DIV_256	515.6 kHz
512	SD_DIV_512	257.8 kHz

Notes When the argument clock is SD_CLK_400KHZ, the lower limit value of SDCLK is 100 kHz.
 The clock frequency dividing ratio that the clock supplied to the card exceeds the clock dividing frequency of the argument clock should not be the return value.

```
Creation      /* Clock frequency dividing ratio acquisition example (operation clock:
example      132 MHz) */
             #include "r_sdif.h"

             uint32_t sddev_get_clockdiv(int32_t sd_port, int32_t clock)
             {
                 uint32_t div;

                 switch(clock)
                 {
                     case SD_CLK_50MHZ:
                         div = SD_DIV_4;          /* 132 MHz/4 = 33 MHz      */
                         break;
                     case SD_CLK_25MHZ:
                         div = SD_DIV_8;          /* 132 MHz/8 = 16.5 MHz    */
                         break;
                     case SD_CLK_20MHZ:
                         div = SD_DIV_8;          /* 132 MHz/8 = 16.5 MHz    */
                         break;
                     case SD_CLK_10MHZ:
                         div = SD_DIV_16;         /* 132 MHz/16 = 8.3 MHz    */
                         break;
                     case SD_CLK_5MHZ:
                         div = SD_DIV_32;         /* 132 MHz/32 = 4.1 MHz    */
                         break;
                     case SD_CLK_1MHZ:
                         div = SD_DIV_256;        /* 132 MHz/256 = 515.6 kHz */
                         break;
                     case SD_CLK_400KHZ:
                         div = SD_DIV_512;        /* 132 MHz/512 = 257.8 kHz */
                         break;
                     default:
                         div = SD_DIV_512;        /* 132 MHz/512 = 257.8 kHz */
                         break;
                 }
                 return div;
             }
```

4.3.8 sddev_set_port

sddev_set_port

Target CPU Interface Function

Port setting for card

```
Format      #include "r_sdif.h"
            int32_t sddev_set_port(int32_t sd_port, int32_t mode);
            int32_t sd_port      | SDHI channel number (0 or 1)
            int32_t mode         | Set port mode
                                SD_PORT_SERIAL: Serial port setting
                                SD_PORT_PARALLEL: Parallel port setting
```

```
Return      SD_OK           : Normal end
values      SD_ERR          : Error end
```

Description No processing needs to be implemented. The return value should always be SD_OK.

4.3.9 sddev_int_wait**sddev_int_wait**

Target CPU Interface Function

Card interrupt standby

Format `#include "r_sdif.h"`

```
int32_t sddev_int_wait(int32_t sd_port, int32_t time);
      int32_t sd_port      |   SDHI channel number (0 or 1)
      int32_t time         |   Time-out time (msec)
```

Return `SD_OK` : Normal end
 values `SD_ERR` : Error end

Description The interrupt waiting process at the protocol communication with the card is executed. When the interrupt request can be confirmed, the `SD_OK` is returned. When the interrupt request cannot be detected in the time-out time of time time, the `SD_ERR` is returned.

The interrupt waiting process is mounted with either process by the software polling or processing that uses interrupt.

The interrupt request can be confirmed by using the `sd_check_int` function. Confirm if the request has been generated by calling the `sd_check_int` function in this function.

Notes The protocol communication interrupt might have been generated before this function is called. Note that the interrupt processing and the interrupt callback function might have been called before this function is called especially when the hardware interrupt is used. When the `sd_mount` function is executed, this function is used for the time measuring process. Therefore, the `SD_ERR` must be returned after the time more than the millisecond of time-out time time has passed when the interrupt cannot be confirmed.

Creation
example

```
/* Card interrupt standby example */
#include "r_sdif.h"

int32_t sddev_int_wait(int32_t sd_port, int32_t time)
{
    /* Interrupt request waiting */
    while(1)
    {
        if(sd_check_int(sd_port) == SD_OK)
        {
            /* There is an interrupt request */
            break;
        }
        /* Time-out processing etc. */
    }
    return SD_OK;
}
```

4.3.10 sddev_loc_cpu

sddev_loc_cpu	Target CPU Interface Function
Card interrupt disable	

Format	#include "r_sdif.h" int32_t sddev_loc_cpu(int32_t sd_port) int32_t sd_port I SDHI channel number (0 or 1)		
Return values	SD_OK	:	Normal end
Description	No processing needs to be implemented. The return value should always be SD_OK.		

4.3.11 sddev_unl_cpu

sddev_unl_cpu	Target CPU Interface Function
Card interrupt enable	

Format	#include "r_sdif.h" int32_t sddev_unl_cpu(int32_t sd_port) int32_t sd_port I SDHI channel number (0 or 1)	
Return values	SD_OK	: Normal end
Description	No processing needs to be implemented. The return value should always be SD_OK.	

4.3.12 sddev_init_dma**sddev_init_dma**

Target CPU Interface Function

Data transfer DMA initialization

Format `#include "r_sdif.h"`
 `int32_t sddev_init_dma(int32_t sd_port, uint32_t buff, int32_t dir);`
 `int32_t sd_port` | SDHI channel number (0 or 1)
 `uint32_t buff` | The first address in buffer to transfer
 `int32_t dir` | Transfer direction
 0: SD_BUF0 register -> buffer
 1: Buffer -> SD_BUF0 register

Return `SD_OK` : Normal end
 values `SD_ERR` : Error end

Description The DMA setting for the card access is executed.
 Table 4.23 lists the DMA controller setting.

Table 4.23 DMA Controller Setting

DMA Controller	Transferring Direction	
	dir = 0	dir = 1
DMAC channel select	SD upstream	SD downstream
DMAC bus width select	64-bit, fixed	
Destination address/source address (8-byte units)	Start address of buffer to be transferred	

Notes When the software transfer is selected in the operation mode of the `sd_mount` function, this function is not used. Therefore it should be mounted as an empty function.

4.3.13 sddev_wait_dma_end

sddev_wait_dma_end

Target CPU Interface Function

Data transfer DMA transfer completion standby

Format `#include "r_sdif.h"`
 `int32_t sddev_wait_dma_end(int32_t sd_port, int32_t cnt);`
 `int32_t sd_port` | SDHI channel number (0 or 1)
 `int32_t cnt` | DMA transfer byte count

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description Waits for the DMA transfer completion set by the `sddev_init_dma` function.
 The method of the DMA transfer completion waiting depends on the system. It should be
 mounted according to the system; for example, the DMA transfer completion interrupt and
 poling the DMA controller register.
 When the DMA transfer is complete, the `SD_OK` should be returned and the function
 process is terminated.
 It is recommended to set the time-out time corresponding to the system based on the DMA
 transfer byte count `cnt` even though the time-out time is not provided. When the transferring
 of all data is not completed within the time out time, or the transfer process is not completed
 such as suspended the process in the middle of DMA transfer, the `SD_ERR` should be
 returned.

Notes When the software transfer is selected in the operation mode of the `sd_mount` function, this
 function is not used. Therefore it should be mounted as an empty function.

4.3.14 sddev_disable_dma

sddev_disable_dma

Target CPU Interface Function

Data transfer DMA disable

Format	<pre>#include "r_sdif.h" int32_t sddev_disable_dma(int32_t sd_port); int32_t sd_port SDHI channel number (0 or 1)</pre>		
Return values	SD_OK	:	Normal end
	SD_ERR	:	Error end
Description	Disables the DMA for the card access set by the sddev_init_dma function. This function is called after the completion of DMA transfer process by the sddev_wait_dma_end function is confirmed.		
Notes	When the software transfer is selected in the operation mode of the sd_mount function, this function is not used. Therefore it should be mounted as an empty function.		

4.3.15 sddev_reset_dma

sddev_reset_dma

Target CPU Interface Function

Reset of DMA

Format `#include "r_sdif.h"`
 `int32_t sddev_reset_dma(int32_t sd_port);`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description Resets the DMA.

Notes -

4.3.16 sddev_finalize_dma

sddev_finalize_dma

Target CPU Interface Function

Termination of DMA

Format `#include "r_sdif.h"`
 `int32_t sddev_finalize_dma(int32_t sd_port);`
 `int32_t sd_port` **I** SDHI channel number (0 or 1)

Return `SD_OK` : Normal end
values `SD_ERR` : Error end

Description Terminates the DMA.

Notes -

4.3.17 sddev_cmd0_sdio_mount

sddev_cmd0_sdio_mount

Target CPU Interface Function

Selecting CMD0 issuance when SDIO card is mounted

Format	<pre>#include "r_sdif.h" int32_t sddev_cmd0_sdio_mount(int32_t sd_port); int32_t sd_port I SDHI channel number (0 or 1)</pre>	
Return values	SD_OK	: Issue CMD0 when the SDIO card is mounted
	SD_ERR	: Do not issue CMD0 when the SDIO card is mounted
Description	Selects to issue CMD0 or not when the SDIO card is mounted.	
Notes	-	

4.3.18 sddev_cmd8_sdio_mount

sddev_cmd8_sdio_mount

Target CPU Interface Function

Selecting CMD8 issuance when SDIO card is mounted

Format	<pre>#include "r_sdif.h" int32_t sddev_cmd8_sdio_mount(int32_t sd_port); int32_t sd_port I SDHI channel number (0 or 1)</pre>	
Return values	SD_OK	: Issue CMD8 when the SDIO card is mounted
	SD_ERR	: Do not issue CMD8 when the SDIO card is mounted
Description	Selects to issue CMD8 or not when the SDIO card is mounted.	
Notes	-	

4.4 Device Driver Functions

This section describes how to integrate the SD driver as a collection of device driver functions that are called by FatFs (the generic FAT filesystem). Check the FatFs specifications, and make changes to match the system if necessary.

The device driver function should be made according to the specification of filesystem used when the SD driver is embedded in the other filesystem.

Table 4.24 lists the device driver functions.

Table 4.24 Device Driver Functions

Device Driver Function Name	Function Outline
disk_status	Device status acquisition
disk_initialize	Device initialization
disk_read	Reading sector data (logical sector units)
disk_write	Writing sector data (logical sector units)
disk_ioctl	Control of other device
get_fattime	Date and time acquisition

4.4.1 disk_status

disk_status		Device Driver Function
Device status acquisition		

Format	#include "diskio.h"	
	DSTATUS disk_status (BYTE pdrv)	
	BYTE pdrv	Physical drive number (0 to 9)
Return values	STA_NOINIT	: Flag indicating that device is not initialized
	STA_NODISK	: Flag indicating that media is not present
	STA_PROTECT	: Flag indicating that media is write protected

Description The sample program returns the device status (STA_NODISK, STA_NOINIT, or STA_PROTECT).
If the setting of pdrv is 2 or greater, STA_NODISK | STA_NOINIT is returned.

Creation example

```
/* Device status acquisition example */
#include "diskio.h"

DSTATUS disk_status (BYTE pdrv)
{
    DSTATUS ret;
    int32_t chk;
    ret = 0;

    /* Confirms that physical drive number is valid */
    if (pdrv > 1)
    {
        ret = (STA_NODISK | STA_NOINIT);
    }
    else
    {
        /* Confirms that card is inserted */
        chk = sd_check_media((int32_t)pdrv);
        if (chk != SD_OK)
        {
            ret = (STA_NODISK | STA_NOINIT); /* Media not present */
        }
        /* Checks whether or not card type information has already been
        acquired */
        else if (sd_info[pdrv].type == SD_MEDIA_UNKNOWN)
        {
            ret = STA_NOINIT; /* Device not initialized */
        }
        /* Checks whether or not card is write protected */
        else if (sd_info[pdrv].iswp != SD_WP_OFF)
        {
            ret = STA_PROTECT; /* Media is write protected */
        }
        else
        {
            ret = 0;
        }
    }
    return ret;
}
```

4.4.2 disk_initialize**disk_initialize**

Device Driver Function

Device initialization

Format `#include "diskio.h"`
 `DSTATUS disk_initialize (BYTE pdrv)`
 `BYTE pdrv` `I` Physical drive number (0 to 9)

Return values `STA_NOINIT` : Flag indicating that device is not initialized
 `STA_NODISK` : Flag indicating that media is not present
 `STA_PROTECT` : Flag indicating that media is write protected

Description The sample program initializes the driver and mounts the card.

This function is managed by FatFs. It is called as needed by the auto-mounting operation.
 This function cannot be called by an application.
 If reinitialization is necessary, use the API function (f_mount()) of FatFs.

Creation example `/* Device initialization example */`
 `#include "diskio.h"`

```

/* Sets the start address of the sampling clock controller */
#define SDCFG_IP0_BASE      (0xE8227000uL)    /* Channel 0 */
#define SDCFG_IP1_BASE      (0xE8229000uL)    /* Channel 1 */

/* SD driver buffer area size */
#define SD_RW_BUFF_SIZE     (1 * 1024)

typedef struct
{
    uint16_t type;
    int32_t iswp;
} SD_INFO;

/* Defines SD driver work area */
uint32_t SDTestWork[2][SD_SIZE_OF_INIT/sizeof(uint32_t)];

/* Defines SD driver buffer area */
uint32_t test_sd_rw_buff[2][SD_RW_BUFF_SIZE/sizeof(uint32_t)];

SD_INFO sd_info[2] =
{
    { SD_MEDIA_UNKNOWN, SD_WP_OFF }
    , { SD_MEDIA_UNKNOWN, SD_WP_OFF }
};

static const uint32_t sd_base_addr[2] =
{
    SDCFG_IP0_BASE
    , SDCFG_IP1_BASE
};

DSTATUS disk_initialize (BYTE pdrv)
{
    DSTATUS      ret;

```

```
int32_t      chk;
uint16_t     type;

ret = (STA_NODISK | STA_NOINIT);

/* Confirms that physical drive number is valid */
if (pdrv > 1)
{
    return ret;
}

/* Initializes card information */
sd_info[pdrv].type = SD_MEDIA_UNKNOWN;
sd_info[pdrv].iswp = SD_WP_OFF;

/* Initializes SD driver */
chk = sd_init((int32_t)pdrv
              ,sd_base_addr[pdrv]
              ,&SDTestWork[pdrv][0]
              ,SD_CD_SOCKET);
if (chk != SD_OK)
{
    return ret;
}

/* Confirms that card is inserted */
chk = sd_check_media((int32_t)pdrv);
if (chk != SD_OK)
{
    return ret;
}
else
{
    /* Card is inserted */
    ret &= ~STA_NODISK;

    /* Sets card swapping interrupt */
    chk = sd_cd_int((int32_t)pdrv, SD_CD_INT_ENABLE, NULL);
    if (chk != SD_OK)
    {
        return ret;
    }

    /* Sets SD driver buffer */
    chk = sd_set_buffer((int32_t)pdrv
                       ,&test_sd_rw_buff[pdrv][0]
                       ,SD_RW_BUFF_SIZE);
    if (chk != SD_OK)
    {
        return ret;
    }

    /* Mounts card with settings for hardware interrupt, DMA transfer
       (64-bit, fixed), default-Speed card support, High-Capacity card
       support, and eXtended-Capacity card support */
    chk = sd_mount((int32_t)pdrv
                  ,SD_MODE_HWINT|SD_MODE_DMA|SD_MODE_DS|SD_MODE_VER2X
                  ,SD_VOLT_3_3);
}
```

```
    if (chk != SD_OK)
    {
        return ret;
    }

    /* Gets the card type and operating mode */
    chk = sd_get_type((int32_t)pdrv, &type, NULL, NULL);
    if (chk != SD_OK)
    {
        return ret;
    }
    sd_info[pdrv].type = type;

    /* Gets write protection information */
    chk = sd_iswp((int32_t)pdrv);
    if (chk == SD_ERR)
    {
        return ret;
    }
    sd_info[pdrv].iswp = chk;

    /* Initialization end */
    ret &= ~STA_NOINIT;

    /* Checks whether or not card is write protected */
    if (sd_info[pdrv].iswp != SD_WP_OFF)
    {
        ret |= STA_PROTECT; /* Media is write protected */
    }
}
return ret;
}
```

4.4.3 disk_read**disk_read**

Device Driver Function

Reading sector data (logical sector units)

Format	<pre>#include "diskio.h" DRESULT disk_read (BYTE pdrv, BYTE* buff, DWORD sector, UINT count) BYTE pdrv I Physical drive number (0 to 9) BYTE* buff O Pointer to read buffer DWORD sector I Read start sector number UINT count I Read sector count (1 to 128)</pre>		
Return values	RES_OK	:	Normal end
	RES_ERROR	:	Error during read
	RES_PARERR	:	Invalid command
	RES_NOTRDY	:	Storage device not in operable state (Not used in sample program.)
Description	<p>Reads sector data from the drive specified by physical drive number pdrv. Sector data equal to the read sector count is read, starting from the sector specified by start sector number sector, and the data is stored in the area indicated by pointer buff.</p>		
Creation example	<pre>/* Sector data read example */ #include "diskio.h" DRESULT disk_read (BYTE pdrv, BYTE *buff, DWORD sector, UINT count) { DRESULT ret; int32_t chk; ret = RES_PARERR; /* Confirms that physical drive number is valid */ if (pdrv > 1) { return ret; } /* Reads sector data from card */ chk = sd_read_sect(pdrv, buff, sector, count); if (chk == SD_OK) { ret = RES_OK; } else { ret = RES_ERROR; } return ret; }</pre>		

4.4.4 disk_write**disk_write**

Device Driver Function

Writing sector data (logical sector units)

Format	#include "diskio.h"		
	DRESULT disk_write (BYTE pdrv, const BYTE* buff, DWORD sector, UINT count)		
	BYTE pdrv		Physical drive number (0 to 9)
	const BYTE* buff		Pointer to data to be written
	DWORD sector		Write start sector number
	UINT count		Write sector count (1 to 128)
Return values	RES_OK	:	Normal end
	RES_ERROR	:	Error during write
	RES_PARERR	:	Invalid command
	RES_NOTRDY	:	Storage device not in operable state (Not used in sample program.)

Description Writes sector data to the drive specified by physical drive number pdrv. The data indicated by pointer buff, equal to the write sector count, is written, starting from the sector specified by start sector number sector.

Creation example

```

/* Sector data write example */
#include "diskio.h"

DRESULT disk_write (BYTE pdrv, const BYTE *buff, DWORD sector
                    ,UINT count)
{
    DRESULT ret;
    int32_t chk;

    ret = RES_PARERR;
    /* Confirms that physical drive number is valid */
    if (pdrv > 1)
    {
        return ret;
    }
    /* Writes sector data to card */
    chk = sd_write_sect(pdrv
                        ,(uint8_t *)buff
                        ,sector
                        ,count
                        ,SD_WRITE_OVERWRITE);

    if (chk == SD_OK)
    {
        ret = RES_OK;
    }
    else
    {
        ret = RES_ERROR;
    }
    return ret;
}

```

4.4.5 disk_ioctl

disk_ioctl

Device Driver Function

Control of other device

Format `#include "diskio.h"``DRESULT disk_ioctl (BYTE pdrv, BYTE cmd, void* buff)``BYTE pdrv | Physical drive number (0 to 9)``BYTE cmd | Control command``void* buff | Data delivery buffer`Return `RES_OK` : Normal endvalues `RES_ERROR` : Error occurred (Not used in sample program.)`RES_PARERR` : Invalid command (Not used in sample program.)`RES_NOTRDY` : Storage device not in operable state (Not used in sample program.)Description In the sample program no processing is performed for all commands, and `RES_OK` is returned.Creation `/* Other device control example */`example `#include "diskio.h"``DRESULT disk_ioctl (BYTE pdrv, BYTE cmd, void *buff)``{``return RES_OK;``}`

4.4.6 get_fattime

get_fattime

Device Driver Function

Date and time acquisition

Format `#include "diskio.h"`
 `DWORD get_fattime (void)`

Return Date and time
values

Description Returns the current local time packed as the value of DWORD.
 The bit field is as follows:
 Bits 31 to 25: Set to a value from 0 to 127, indicating the year, with 1980 as the start point (year 0).
 Bits 24 to 21: Set to a value from 1 to 12, indicating the month.
 Bits 20 to 16: Set to a value from 1 to 31, indicating the day.
 Bits 15 to 11: Set to a value from 0 to 23, indicating the hour
 Bits 10 to 5: Set to a value from 0 to 59, indicating the minute
 Bits 4 to 0: Set to a value from 0 to 29, indicating the second / 2

In the sample program no date and time information is set, and 0x00000000 is returned.

Creation `/* Date and time acquisition example */`
example `#include "diskio.h"`

 `DWORD get_fattime(void)`
 `{`
 `return 0x00000000;`
 `}`

5. Configuration Options

The SD driver configuration options are shown in Table 5.1.

Table 5.1 Configuration Options

Definition	Description
SD_CD_ENABLED	SD card detection option SD card detection is enabled.
SD_CD_DISABLED	SD card detection option SD card detection is disabled. When SD card detection is disabled, the status is always "loading."
SD_WP_ENABLED	Write protection signal detection option Write protection signal detection is enabled.
SD_WP_DISABLED	Write protection signal detection option Write protection signal detection is disabled. When write protection signal detection is disabled, the status is always "write protection signal off."
SD_CB_UNUSED	SD card detection callback function setting SD card detection callback function setting is unused.
SD_CB_USED	SD card detection callback function setting SD card detection callback function setting is used.

6. Restrictions for Application Making

This chapter explains various notes for SD driver.

6.1 Notes for Using SD Driver

The notes when the application program is made by using the SD driver are shown as follows.

(a) Reserved Word

In the SD driver, the following keywords are the reserved words when an application is made with C language.

“Keyword starting with the character string of sd_”

“Keyword starting with the character string of _sd_”

When the application program is made with the assembly language, the following keywords are the reserved words.

“Keyword starting with the character string of _sd_”

“Keyword starting with the character string of __sd_”

(b) The Setting Rule for Arguments and the Guarantee Rule of Registers

The function provided in this library is made on the assumption that it is called from the application program written by C language. The setting rule for SD driver arguments and the guarantee rule for the registers conform to the setting rules and the guarantee rules of cross tool kit. For details, refer to the related manuals.

(c) Notes for Using OS

The SD driver doesn't guarantee the reentrant structure (the structure that can be used from two or more programs (tasks) at the same time). Therefore, the application using the real-time OS, the exclusion control is necessary by using the OS function.

(d) Notes for Using the Interrupt Callback Function

The interrupt callback function is called as the subroutine of the interrupt handler. The function that can be used in the interrupt handler should be used in the interrupt callback function. All the SD driver library functions except the sd_int_handler function cannot be used in the interrupt handler.

7. Sample Program

This chapter provides information on installing the sample program.

7.1 Function Overview

The sample program includes 31 sample processing routines that use the library functions of the RZ/A2 SD driver to implement basic control of the SD card. Please refer to chapter 7.6 for how to execute the sample program.

This sample program consists of three Sample Processing Routines. The overview of the each Sample Processing Routines is shown in Table 7.1. And the list of the Sample Processing Routines is shown in Table 7.2 to Table 7.4.

Table 7.1 Overview of Sample Processing Routines

Sample Processing Routine	Overview	Detail
FatFs Sample Processing Routines	They are a set of commands used to access the SD memory card in conjunction with FatFs.	Refer to Table 7.2
SD Sample Processing Routines	They are a set of commands to access the SD memory card or the SDIO card.	Refer to Table 7.3
SDIO Sample Processing Routines	They are a set of commands to access the SDIO card.	Refer to Table 7.4

Table 7.2 FatFs Sample Processing Routines

Sample Processing Routine	Describing of Sample Processing	Command
Help	Displays a list of the available FatFs sample commands.	HELP
List display	Displays a list of the files and subdirectories in the specified directory.	DIR (directory name)
Display file contents	Displays the contents of the specified file. Displays an error if the specified file does not exist.	TYPE (file name)
Write file	Writes the character string "Renesas FAT/exFAT sample." to the specified file. If the specified file does not exist, a new file is created and the character string is written to it.	WRITE (file name)
Create new file	Creates a new file with the specified name. Displays an error if a file with the same name already exists.	CREATE (file name)
Delete file	Deletes the specified file.	DEL (file name)
Create new directory	Creates a new directory with the specified name. Displays an error if a directory with the same name already exists.	MKDIR (directory name)
Delete directory	Deletes the specified directory. Displays an error if any files, etc., exist in the directory.	RMDIR (directory name)

Note1: The set of commands are valid only if the `SELECT_SD_SDIO_SAMPLE_PROCESSING_ROUTINES` macro is not defined.

Note2: The SD memory card has to be formatted before the command can be executed.

Table 7.3 SD Sample Processing Routines

Sample Processing Routine	Descripting of Sample Processing	Command
SD/SDIO Help	Displays a list of the available SD/SDIO sample commands.	SDHELP
Driver initialization	Initialize the SD/SDIO driver and the SD HOST IP.	INIT
Finalize the driver.	Finalize the SD driver and the SD HOST IP.	FINAL
Mount	Mount the SD card.	IOATT
Unmount	Unmount the SD card.	IODET
Format	Format the SD card.	FORMAT
Type information display	Display the SD card type information.	SDTYPE
Write protect information display	Display the SD card write protect information.	WP
Register display	Display the SD card register.	REG
RCA register display	Display the SD card RCA register (OCR, CID, CSD, DSR and SCR).	RCA
SDSTATUS register display	Display the SD card SDSTATUS register.	SDSTATUS
Card speed information display	Display the SD card speed information.	SPEED
SECCNT set	Set SECCNT value.	SSEC
SECCNT value display	Display SECCNT value.	GSEC
Size information display	Display the SD card size information.	SIZE
IP version information display	Display the SD/SDIO host IP version information.	IPVER
Lock	Lock the SD card operation.	LOCK
Unlock	Unlock the SD card operation.	UNLOCK
Sector Read	Read the SD card sector.	READ
Sector Write	Write the SD card sector.	WRITE

Note1: The set of commands are valid only if the SELECT_SD_SDIO_SAMPLE_PROCESSING_ROUTINES macro is defined.

Note2: If customers want to use sample processing routines that is not on the table, do so under their own responsibility

Table 7.4 SDIO Sample Processing Routines

Sample Processing Routine	Descripting of Sample Processing	Command
SDIO read (CMD52)	Read the SDIO with the CMD52.	IOREAD_D
SDIO write (CMD52)	Write the SDIO with CMD52.	IOWRITE_D
SDIO read (CMD53)	Read the SDIO with the CMD53.	IOREAD
SDIO write (CMD53)	Write the SDIO with CMD53.	IOWRITE

Note1: The set of commands are valid only if the SELECT_SD_SDIO_SAMPLE_PROCESSING_ROUTINES macro is defined.

Note2: If customers want to use sample processing routines that is not on the table, do so under their own responsibility.

7.2 Operating Environment

Figure 7.1 shows an example operating environment for the sample program.

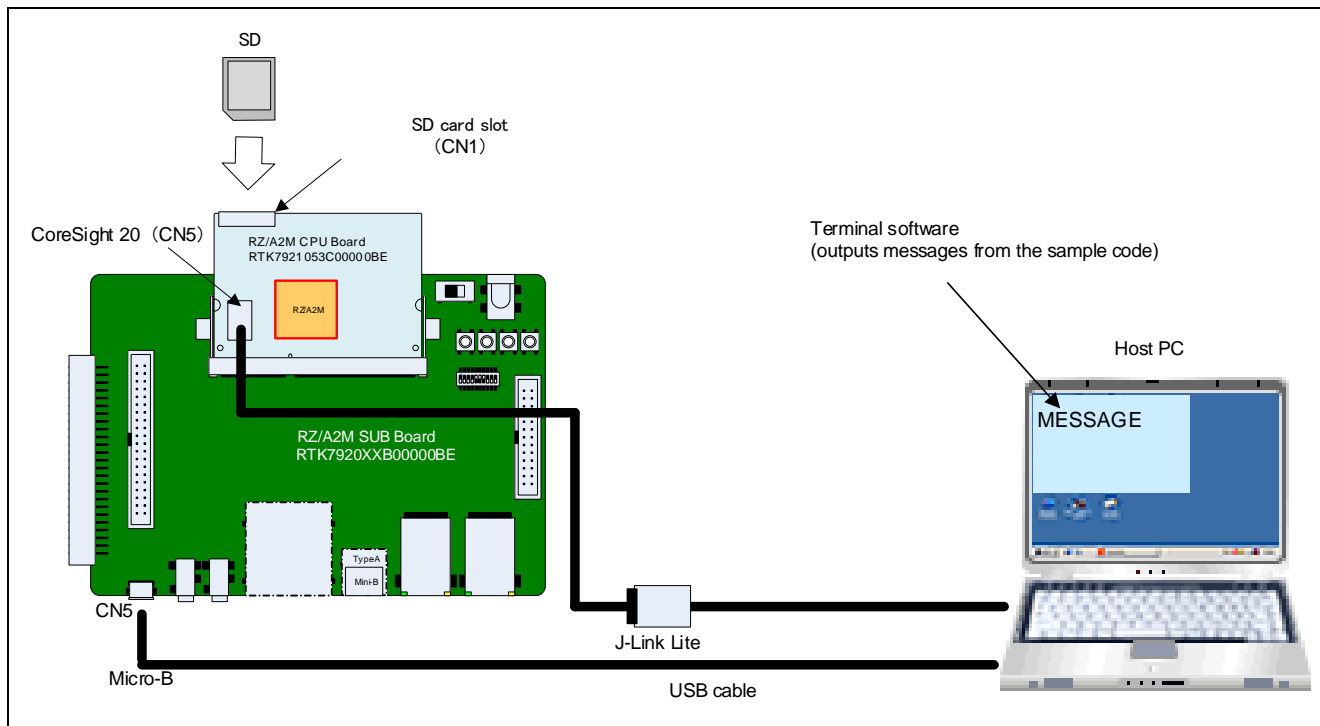


Figure 7.1 Example Operating Environment

7.3 Confirmed Operating Conditions

Table 7.5 lists the confirmed operating conditions of the sample program.

Table 7.5 Confirmed Operating Conditions

item	Contents
Microcomputer used	RZ/A2M
Operating frequency (Note)	CPU Clock ($I\phi$) : 528MHz Image processing clock ($G\phi$) : 264MHz Internal Bus Clock ($B\phi$) : 132MHz Peripheral Clock 1 ($P1\phi$) : 66MHz Peripheral Clock 0 ($P0\phi$) : 33MHz QSPI0_SPCLK : 66MHz CKIO : 132MHz
Operating voltage	Power supply voltage (I/O): 3.3 V Power supply voltage (either 1.8V or 3.3V I/O (PVcc SPI)) : 3.3V Power supply voltage (internal): 1.2 V
Integrated development environment	e2 studio V7.7.0
Emulator	J-Link Lite
C compiler	"GNU Arm Embedded Tool chain 6-2017-q2-update" compiler options(except directory path) Release: -mcpu=cortex-a9 -march=armv7-a -marm -mlittle-endian -mfloat-abi=hard -mfpu=neon -mno-unaligned-access -Os -ffunction-sections -fdata-sections -Wunused -Wuninitialized -Wall -Wextra -Wmissing-declarations -Wconversion -Wpointer-arith -Wpadded -Wshadow -Wlogical-op -Waggregate-return -Wfloat-equal -Wnull-dereference -Wmaybe-uninitialized -Wstack-usage=100 -fabi-version=0 Hardware Debug: -mcpu=cortex-a9 -march=armv7-a -marm -mlittle-endian -mfloat-abi=hard -mfpu=neon -mno-unaligned-access -Og -ffunction-sections -fdata-sections -Wunused -Wuninitialized -Wall -Wextra -Wmissing-declarations -Wconversion -Wpointer-arith -Wpadded -Wshadow -Wlogical-op -Waggregate-return -Wfloat-equal -Wnull-dereference -Wmaybe-uninitialized -g3 -Wstack-usage=100 -fabi-version=0
Operation mode	Boot mode 3 (Serial Flash boot 3.3V)
Terminal software communication settings	<ul style="list-style-type: none"> • Communication speed: 115200bps • Data length: 8 bits • Parity: None • Stop bits: 1 bit • Flow control: None
Board to be used	RZ/A2M CPU board RTK7921053C00000BE RZ/A2M SUB board RTK79210XXB00000BE

Device (functionality to be used on the board)	<ul style="list-style-type: none">Serial flash memory allocated to SPI multi-I/O bus space (channel 0) Manufacturer: Macronix Inc. Model Name : MX25L51245GXDRL78/G1C (Convert between USB communication and serial communication to communicate with the host PC.)LED1SW3SD card Transcend 8 GB (SDHC, class10) FAT filesystem : FAT32
--	---

Note: The operating frequency used in clock mode 1 (Clock input of 24MHz from EXTAL pin)

7.4 Pin Names and Functions

Table 7.6 to Table 7.7 list the pin names and functions used by the sample program.

Table 7.6 Pin Names and Functions 1 (SD/MMC Card Slot on Channel 0)

Pin Name	I/O	Function	Remarks
SD0_CLK	Output	SD clock SD clock output pin	3.3 V, fixed
SD0_CMD	Input/ output	SD command SD command output/ response input signal	3.3 V, fixed
SD0_DAT0	Input/ output	SD data 0 Data [Bit0] signal	3.3 V, fixed
SD0_DAT1	Input/ output	SD data 1 Data [bit 1]/SDIO interrupt signal	3.3 V, fixed
SD0_DAT2	Input/ output	SD data 2 Data [bit 2]/read wait signal	3.3 V, fixed
SD0_DAT3	Input/ output	SD data 3 Data [Bit 3]/card detection signal	3.3 V, fixed
SD0_CD	Input	SD card detection SD card detection input signal	Optional 3.3 V, fixed
SD0_WP	Input	SD write protection SD write protection input signal	Optional 3.3 V, fixed
SD0_RST	Output	SD reset SD reset output signal	3.3 V, fixed
PD_1	Output	SD command/ SD data 0 to 3 output voltage switch	High = 3.3 V, fixed
PJ_1	Input	SW3 key input	-
P6_0	Output	LED1(RED)	-
PC_1	Output	LED1(Yellowish-green)	-

Table 7.7 Pin Names and Functions 2 (SD/MMC Card Slot on Channel 1)

Pin Name	I/O	Function	Remarks
SD1_CLK	Output	SD clock SD clock output pin	3.3 V, fixed
SD1_CMD	Input/ output	SD command SD command output/ response input signal	3.3 V, fixed
SD1_DAT0	Input/ output	SD data 0 Data [Bit0] signal	3.3 V, fixed
SD1_DAT1	Input/ output	SD data 1 Data [bit 1]/SDIO interrupt signal	3.3 V, fixed
SD1_DAT2	Input/ output	SD data 2 Data [bit 2]/read wait signal	3.3 V, fixed
SD1_DAT3	Input/ output	SD data 3 Data [Bit 3]/card detection signal	3.3 V, fixed
SD1_CD	Input	SD card detection SD card detection input signal	Optional 3.3 V, fixed
SD1_WP	Input	SD write protection SD write protection input signal	Optional 3.3 V, fixed

7.5 Memory Size

Table 7.8 lists the amount of memory used by the SD driver.

Table 7.8 SD Driver Memory Usage

ROM (KB)	RAM (KB)	Stack (KB)
38.0	1.0	0.5

7.6 Installation Procedure

The installation procedure for the sample program is described below.

1. Preparing the operating environment
Refer to 7.2, Operating Environment, and prepare the environment in which to run the sample program.
2. How to select the Sample Processing Routine
The FatFs Sample Processing Routines and SD Sample Processing Routines/SDIO Sample Processing Routines cannot be used at same time. Please select one of them by using the `SELECT_SD_SDIO_SAMPLE_PROCESSING_ROUTINES` macro. Please refer to Table 7.9.

Table 7.9 How to select the Sample Processing Routine

Selected Sample Processing Routine	How to select the Sample Processing Routine
FatFs Sample Processing Routines	The <code>SELECT_SD_SDIO_SAMPLE_PROCESSING_ROUTINES</code> macro in <code>application_common.h</code> should not be defined.
SD Sample Processing Routines	The <code>SELECT_SD_SDIO_SAMPLE_PROCESSING_ROUTINES</code> macro in <code>application_common.h</code> should be defined.
SDIO Sample Processing Routines	The <code>SELECT_SD_SDIO_SAMPLE_PROCESSING_ROUTINES</code> macro in <code>application_common.h</code> should be defined.

Please refer to 3 to 11 if you select the FatFs Sample Processing Routines. And, please refer to 3, 4, 12 and 13 if you select the SD Sample Processing Routines/SDIO Sample Processing Routines.

3. Building the sample program
Build the sample program.
4. Launching the debugger
Launch the debugger and load the sample program.
5. Establishing connection with the SD card
By running the sample program, automatically mount the SD card and establish the SD card connection. If the SD card connection has not been established, the LED1 (RED) lights up.
The mounting of the SD card is performed automatically when the power is on or when the card is inserted.

Table 7.10 LED1 lighting pattern

State	LED1 (RED)	LED1 (Yellowish-green)
Normal The SD card connection establishment succeeded. The SD card connection disconnect succeeded, etc.	Off	Off
Error The SD card connection establishment failed. Card removal during the SD card access, etc.	On	Off
Accessing the SD card	Off	Flashing

6. Accessing to the SD card

(1) The OS version

After running a program, a welcome message is displayed. After establishing a connection with the SD card, you can issue the command via the terminal.

Refer to Table 7.2 for a list of available commands. However, issuing the command while accessing the SD card by SW3 Press is invalid.

If no SD card is connected, the LED1 will turn red. To run the simple SD card access routine, SW3 needs to be pressed for more than 1 second.

Table 7.11 SW3 press pattern for OS version

SW3 press pattern	Condition	Sample Processing Routine
SW3 press	SW3 pressed for more than 1 second.	Simple SD card access routine. This will flash the LED1(Yellowish-green) 10 times. If no SD card is connected or card is removed, the LED1 will turn red.

(2) The OS-less version

After running the program, a welcome message is not displayed so you have to start the Sample Processing Routine by using SW3.

After establishing the connection with the SD card, by the SW3 long press, a welcome message is displayed and the terminal command can be accepted. Refer to Table 7.2 for a list of available commands. Thereafter SW3 press is invalid. If you want to enable SW3 again, remove the SD card and reinsert it.

If no SD card is connected, the LED1 will turn red. To run the simple SD card access routine, SW3 needs to be pressed for a time between 1 and 5 seconds.

Table 7.12 SW3 press pattern for OS-less version

SW3 press pattern	Condition	Sample Processing Routine
SW3 short press	SW3 pressed between 1 and 5 seconds.	Simple SD card access routine. This will flash the LED1(Yellowish-green) 10 times. If no SD card is connected or card is removed, the LED1 will turn red.
SW3 long press	SW3 pressed for more than 5 seconds.	The terminal command has transitioned to a valid state. Thereafter subsequent SW3 presses are invalid. For terminal connection settings please see Table 7.5.

In the example shown in the figure, exFAT is enabled. The same applies to the figures that follow.

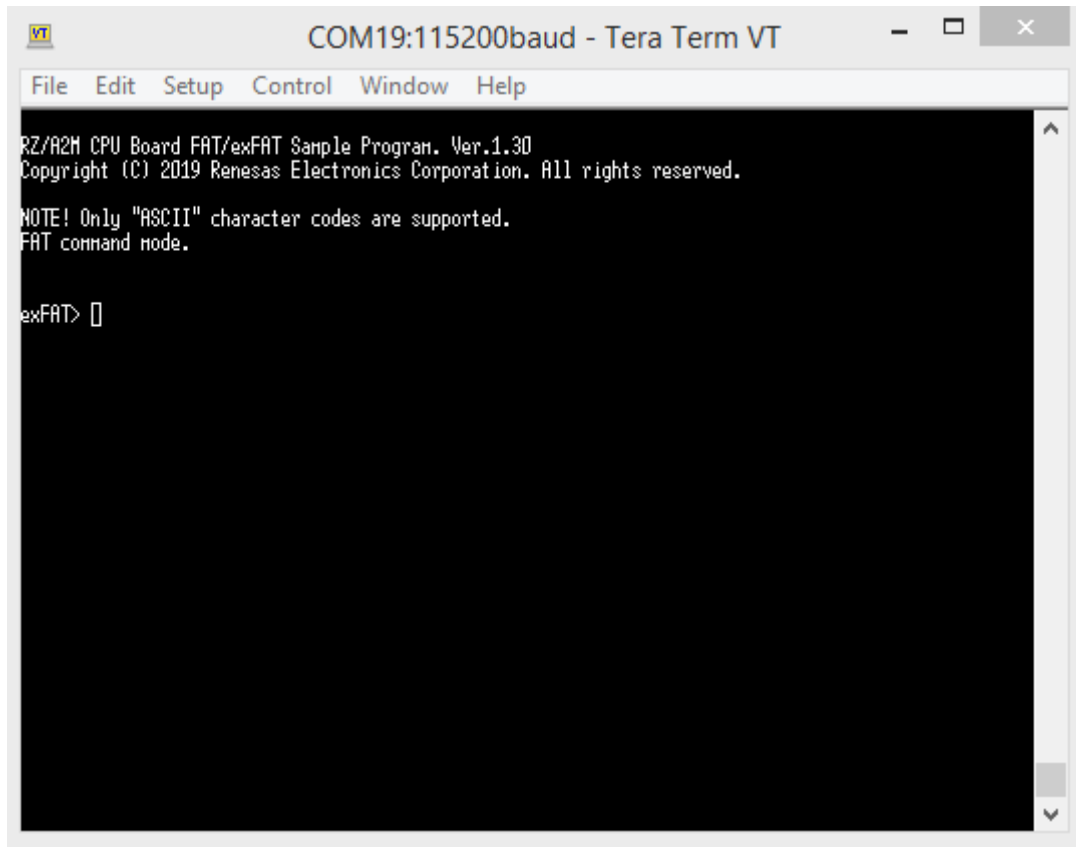


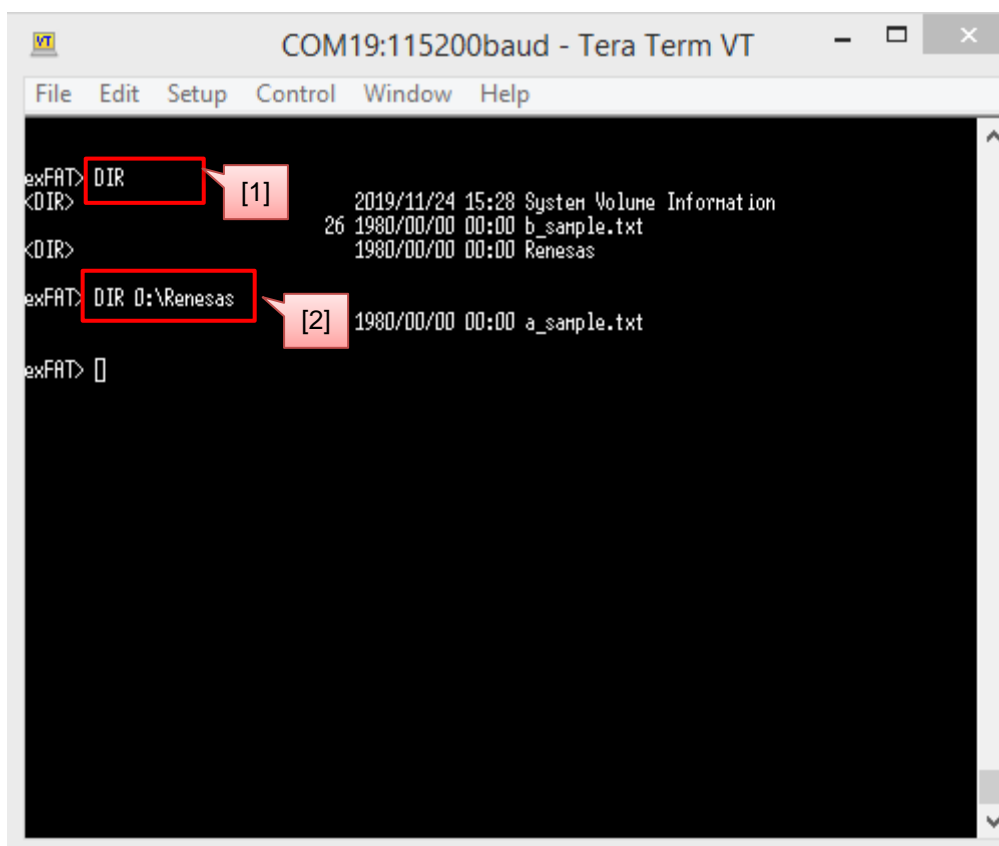
Figure 7.2 Example: Welcome message display

7. Disconnecting from the SD card
By removing the SD card, the SD card will be dismounted automatically and will disconnect the SD card connection.
8. Establishing reconnection with the SD card
Inserting the SD card will automatically re-mount the SD card and establish the SD card connection.

9. Displaying a listing

From the terminal window, issue the command DIR ([1] in the figure (listing of the directory) and [2] in the figure (listing of directory Renesas)) to display listings of the files and subdirectories in the specified directories on the connected SD card.

[2] in the figure is an example of specifying channel 0.



The screenshot shows a terminal window titled "COM19:115200baud - Tera Term VT". The window has a menu bar with "File", "Edit", "Setup", "Control", "Window", and "Help". The terminal output shows the following sequence of commands and results:

```
exFAT> DIR [1]
<DIR> 2019/11/24 15:28 System Volume Information
      26 1980/00/00 00:00 b_sample.txt
      1980/00/00 00:00 Renesas

exFAT> DIR 0:\Renesas [2]
1980/00/00 00:00 a_sample.txt

exFAT> 
```

Red boxes highlight the commands "DIR" and "DIR 0:\Renesas", with callouts [1] and [2] respectively. The output for the first command shows a directory listing with a date, time, and file names. The output for the second command shows a file listing with a date, time, and file name.

Figure 7.3 Example: Issuing the DIR Command

10. Displaying file contents

From the terminal window, issue the command TYPE ([3] in the figure) to display the contents of the specified file on the connected SD card ([4] in the figure, hexadecimal display).

[3] in the figure is an example of specifying channel 0.

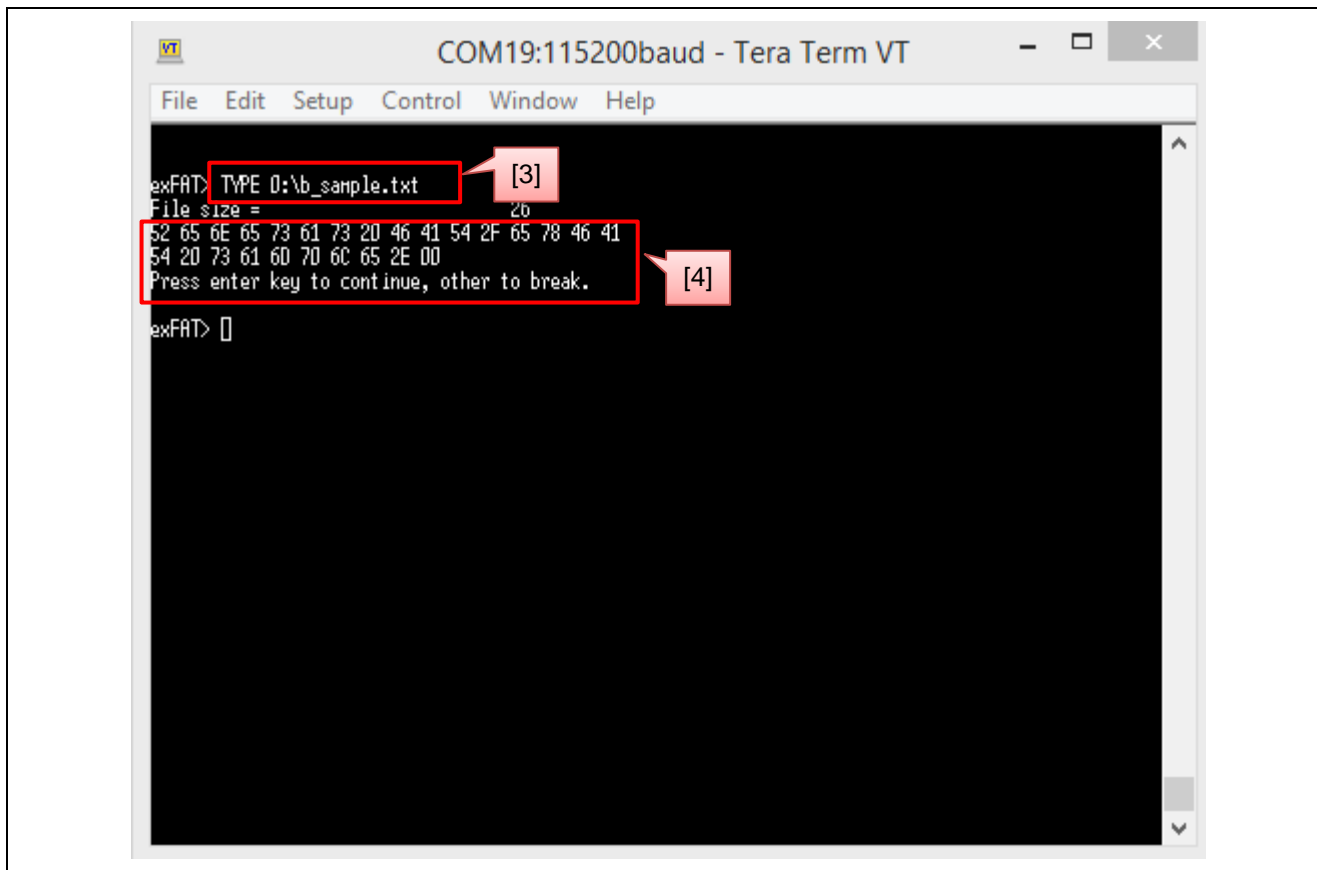
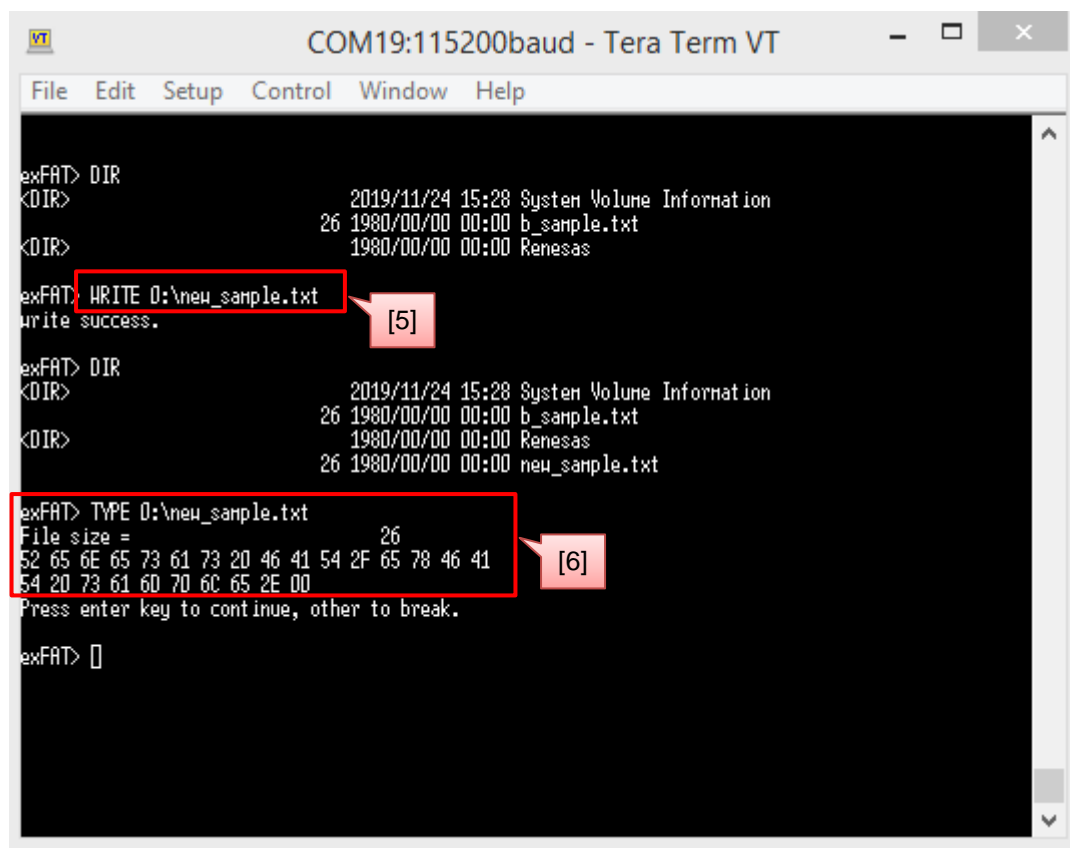


Figure 7.4 Example: Issuing the TYPE Command

11. Writing a file

From the terminal window, issue the command WRITE ([5] in the figure) to write the character string “Renesas FAT/exFAT sample.” to the specified file on the connected SD card ([6] in the figure, the character string written to the card is read and displayed).

[5] and [6] in the figure are an example of specifying channel 0.



```
COM19:115200baud - Tera Term VT
File Edit Setup Control Window Help

exFAT> DIR
<DIR>
2019/11/24 15:28 System Volume Information
26 1980/00/00 00:00 b_sample.txt
<DIR>
1980/00/00 00:00 Renesas

exFAT> WRITE 0:\neu_sample.txt
write success.

exFAT> DIR
2019/11/24 15:28 System Volume Information
26 1980/00/00 00:00 b_sample.txt
<DIR>
1980/00/00 00:00 Renesas
26 1980/00/00 00:00 neu_sample.txt

exFAT> TYPE 0:\neu_sample.txt
File size = 26
52 65 6E 65 73 61 73 20 46 41 54 2F 65 78 46 41
54 20 73 61 60 70 6C 65 2E 00
Press enter key to continue, other to break.

exFAT> 
```

Figure 7.5 Example: Issuing the WRITE Command

12. Establishing the connection with the SD/SDIO card.

The establishment of the connection with the SD card is not performed automatically by inserting the SD card into the SD card slot. It is necessary to execute the following procedures.

From the terminal window, issue the command INIT ([7] in the figure) to initialize the SD/SDIO driver.

[8] in the figure is an example of specifying channel 0.

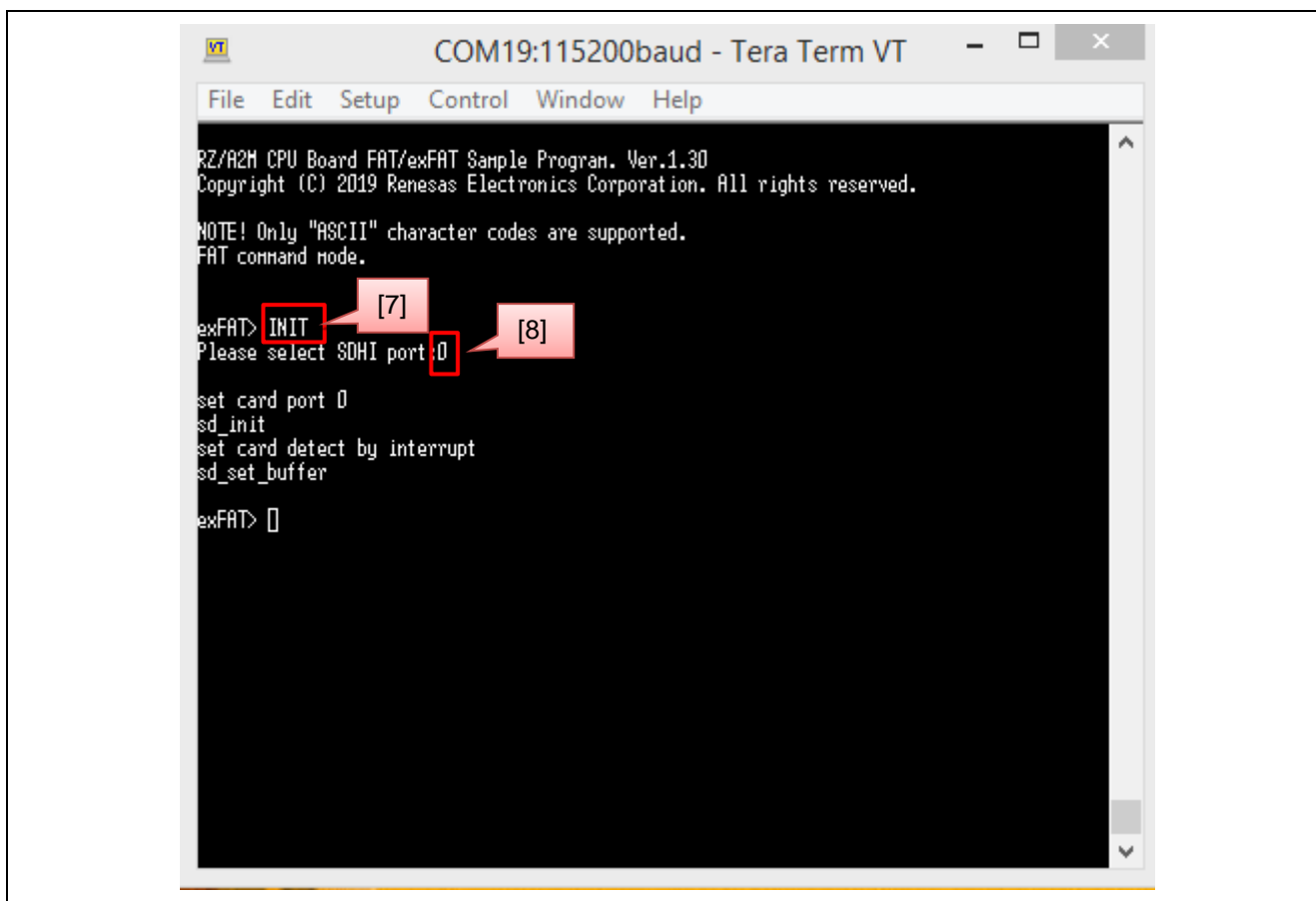


Figure 7.6 Example: Issuing the INIT Command

In addition, from the terminal window, issue the command IOATT ([9] in the figure), and mount the SD/SDIO card to establish a connection to the SD/SDIO card.

[10] in the figure is an example of specifying channel 0.

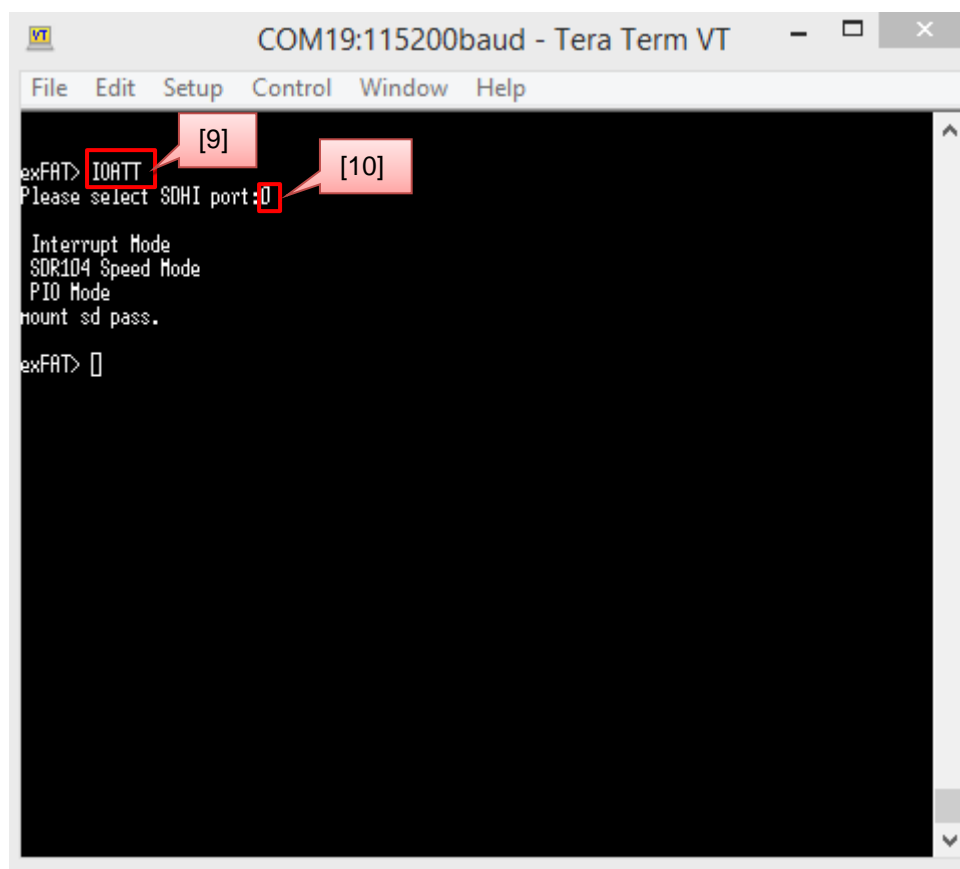


Figure 7.7 Example: Issuing the IOATT Command

13. Disconnecting from the SDIO card

The disconnection of the connection with the SD card is not performed automatically by removing the SD card into the SD card slot. It is necessary to execute the following procedures.

From the terminal window, issue the command IODET ([11] in the figure), unmount the SD/SDIO card, and disconnect from the SD/SDIO card.

[12] in the figure is an example of specifying channel 0.

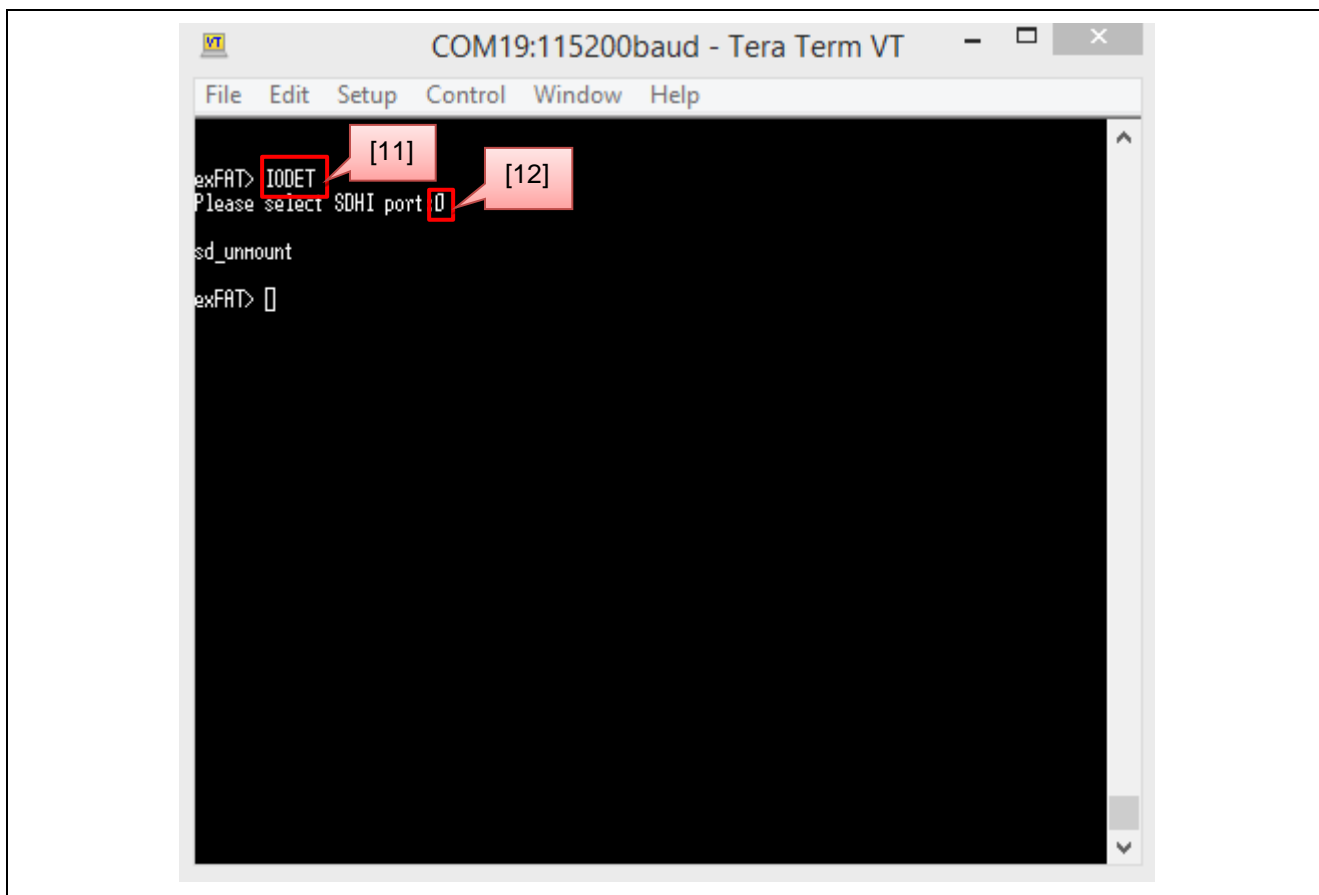


Figure 7.8 Example: Issuing the IODET Command

7.7 Note

Customers adding FatFs to their projects do so under their own responsibility. Make sure to confirm the FatFs licensing terms.

8. Procedure to add the component by Smart Configurator

This chapter provides information on adding the SD driver component by the Smart Configurator.

8.1 Component addition

The procedure to add a component is described below.

1. Select the “Components” tab and press the “Add component” button.

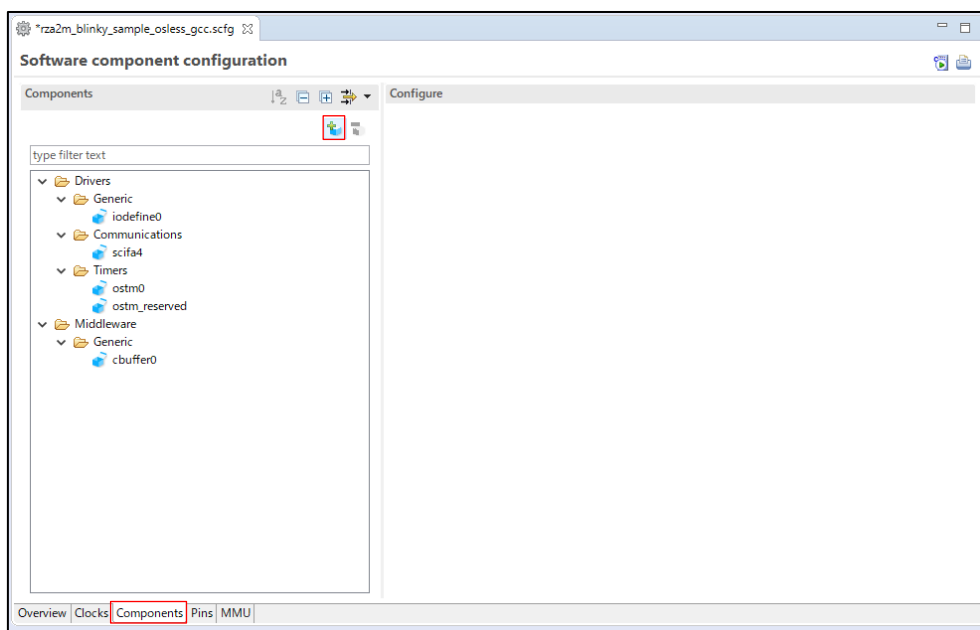


Figure 8.1 Component addition

2. Select the SD driver component “r_sdhi_simplified” and press the “Next >” button.

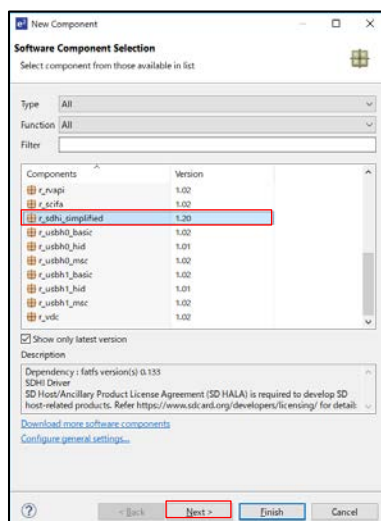


Figure 8.2 Component selection (1/2)

3. Press the “Finish” button. (The figure below is when channel 0 is selected.)

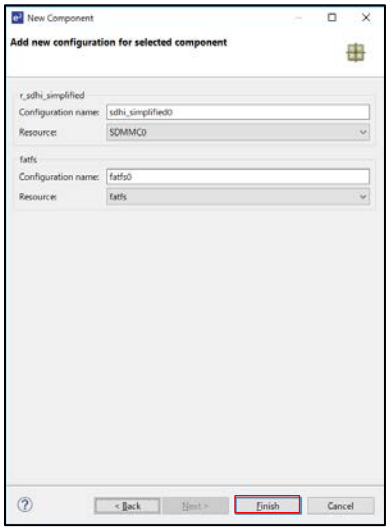


Figure 8.3 Component selection (2/2)

4. The SD driver component “sdhi_simplified0” and the FatFs component “fatfs0” are added.

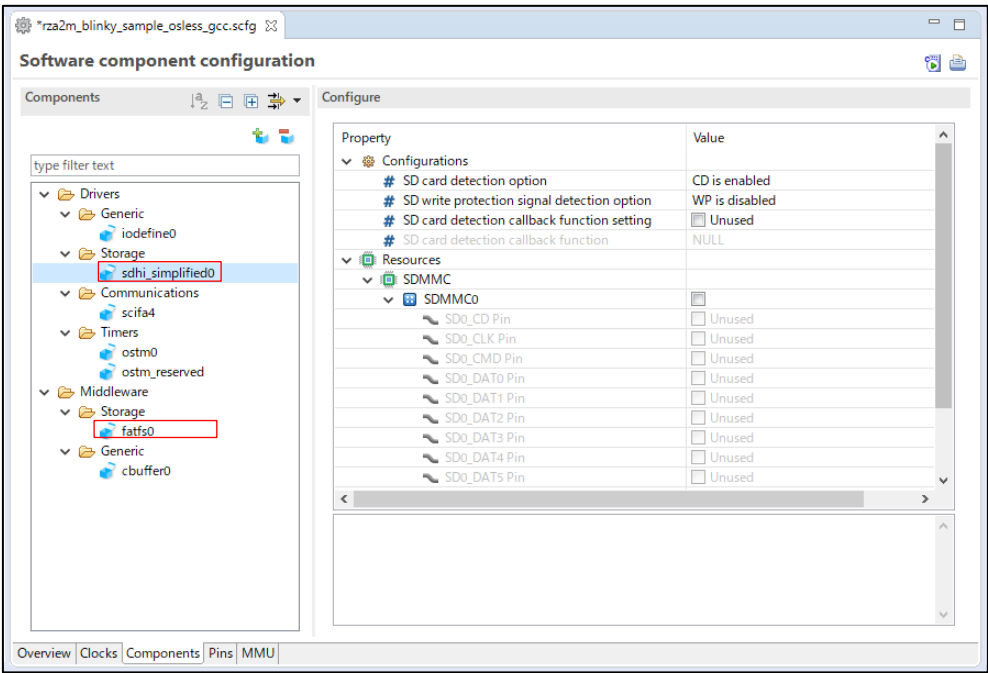


Figure 8.4 Components screen after component addition is completed

8.2 Configuration Settings

The procedure to set the configuration is described below. Refer to “5 Configuration Options” for the configuration options.

The following is an example of a channel 0 setting.

8.2.1 SD card detection option settings

1. Select “sdhi_simplified0” from “Components”, and select “Value” of “Configure”-“Property”-“Configurations”-“SD card detection option”.

Use the default settings (SD card detection is enabled).

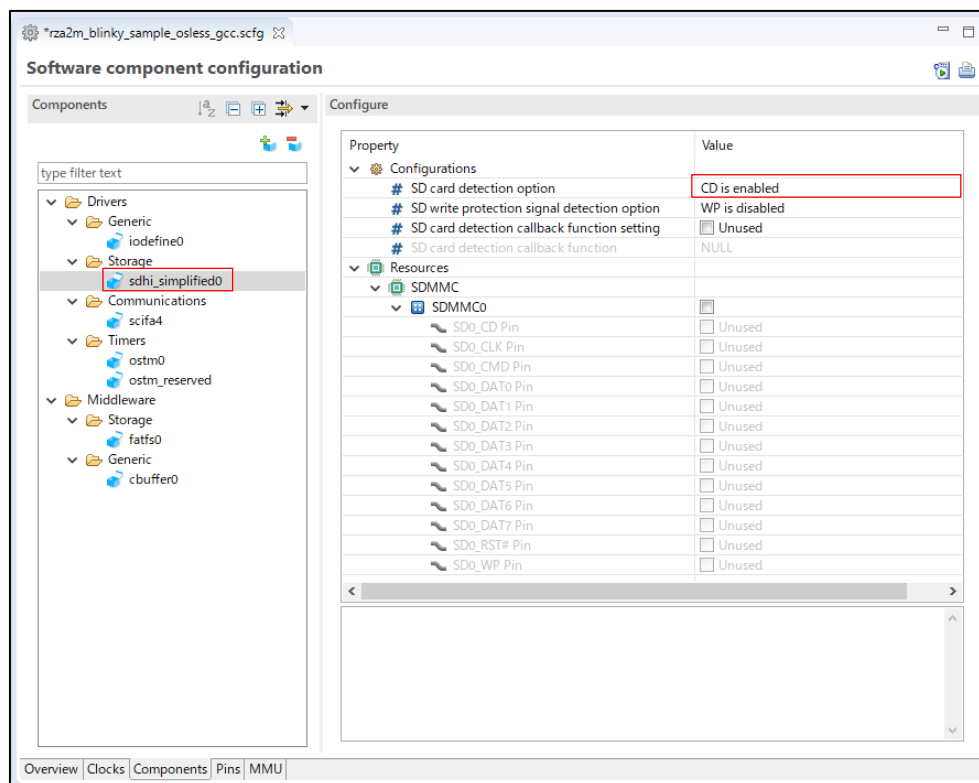


Figure 8.5 SD card detection option settings

8.2.2 SD write protection signal detection option settings

1. Select "sdhi_simplified0" from "Components", and select "Value" of "Configure"- "Property"- "Configurations"- "SD write protection signal detection option".

Use the default settings (write protection signal detection is disabled).

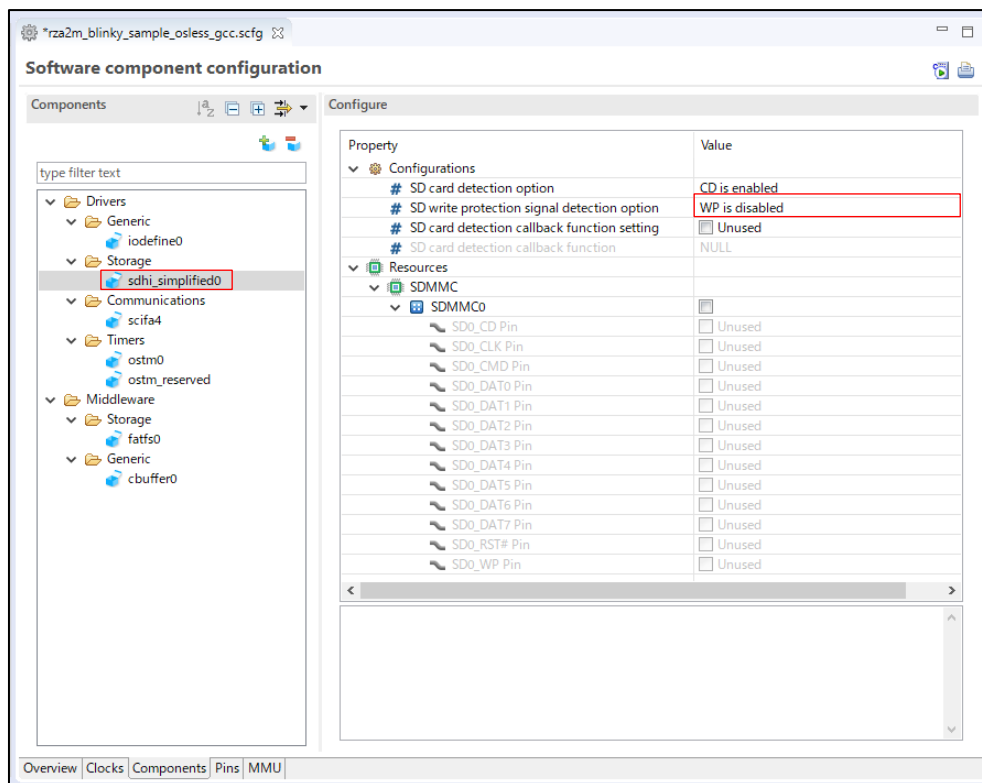


Figure 8.6 SD write protection signal detection option settings

8.2.3 SD card detection callback function settings

1. Select “sdhi_simplified0” from “Components”, and enable “Configure”-“Property”-“Configurations”-“SD card detection callback function setting” check box.

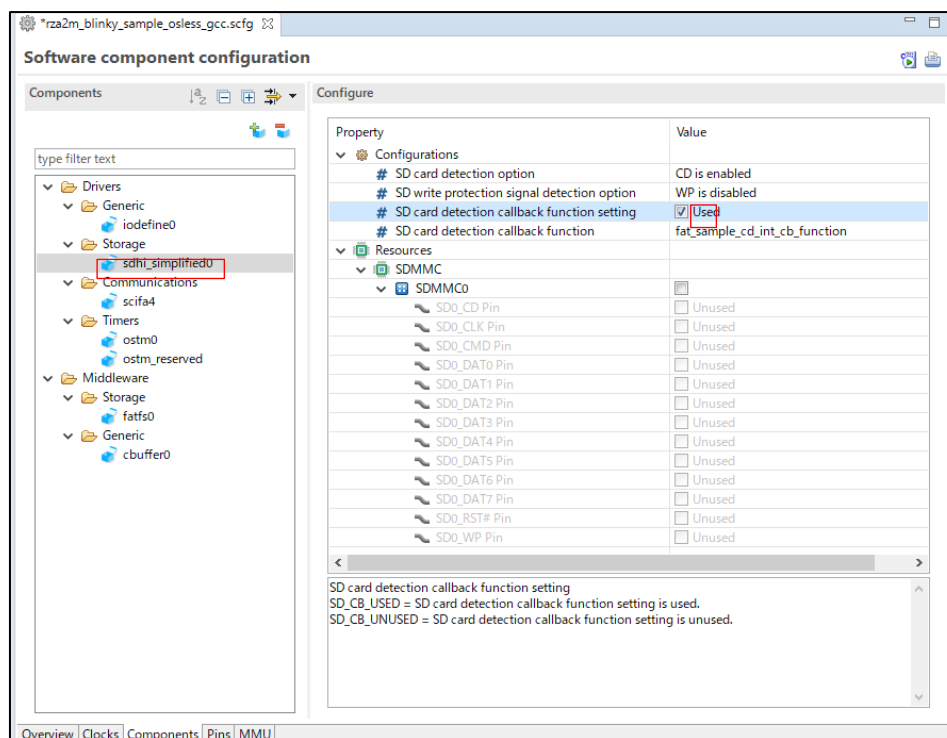


Figure 8.7 SD card detection callback function settings (1/2)

2. Enter the callback function name in “Value” of “SD card detection callback function”.

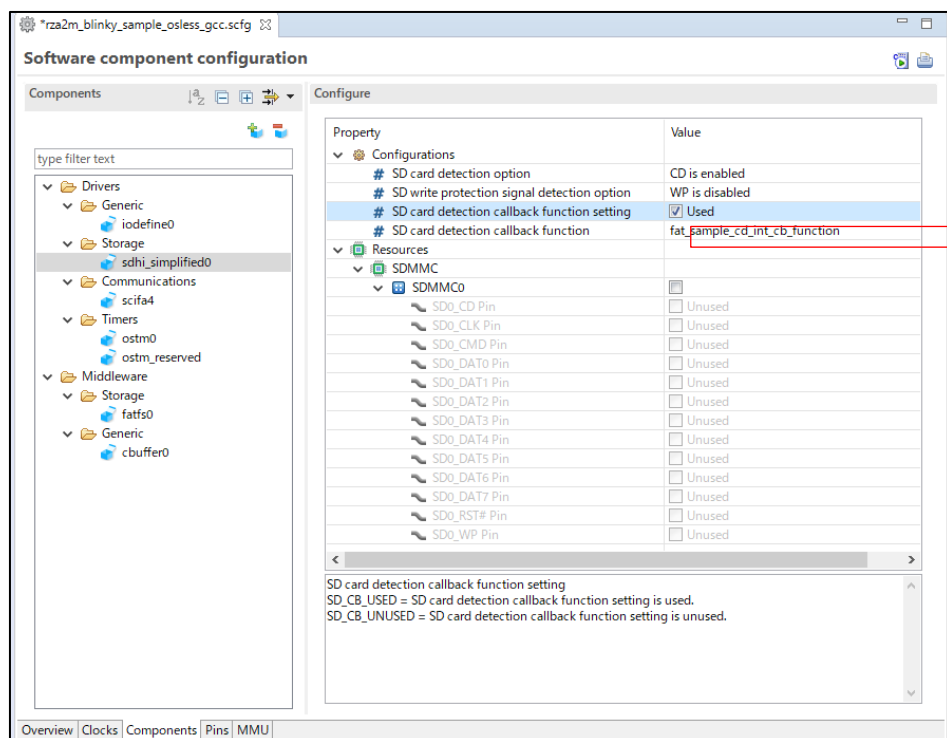


Figure 8.8 SD card detection callback function settings (2/2)

8.3 Pin settings

The procedure to set a pin is described below. Refer to “7.4 Pin Names and Functions” for the pin to be used. The following is an example of a channel 0 setting.

8.3.1 CD pin and WP pin settings

1. Select “sdhi_simplified0” from “Components”, and enable all checkboxes in “Configure”-“Property”-“Resources”-“SDMMC”-“SDMMC0”.

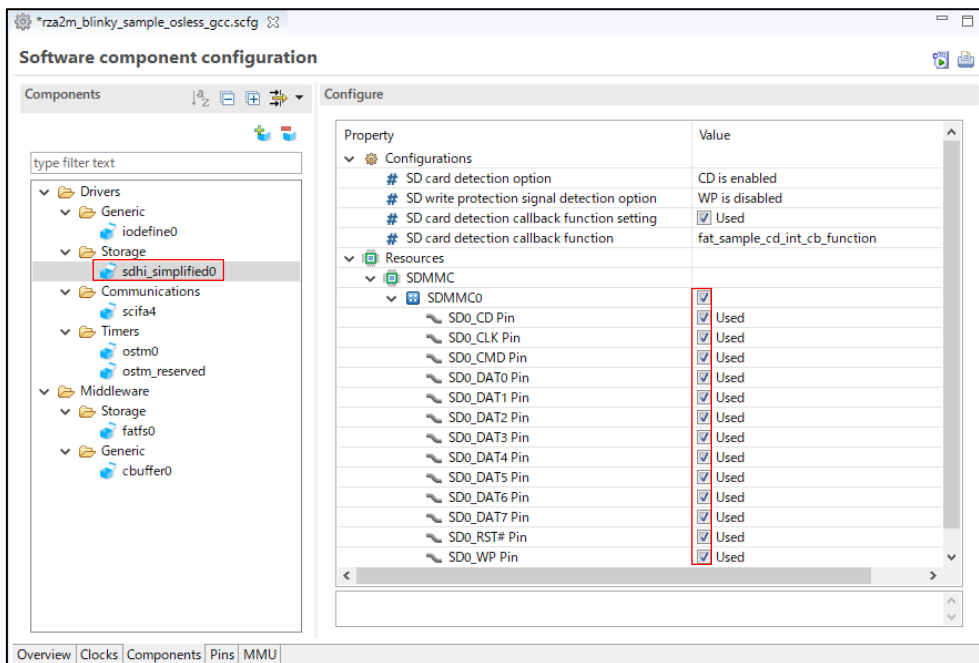


Figure 8.9 SDMMC0 pin selection

2. Select “Pins”-“Pin Function” tab, and select “SD/MMC host interface”-“SDMMC0” from “Hardware Resource”, and select SD0_CD and SD0_WP pin assignments.

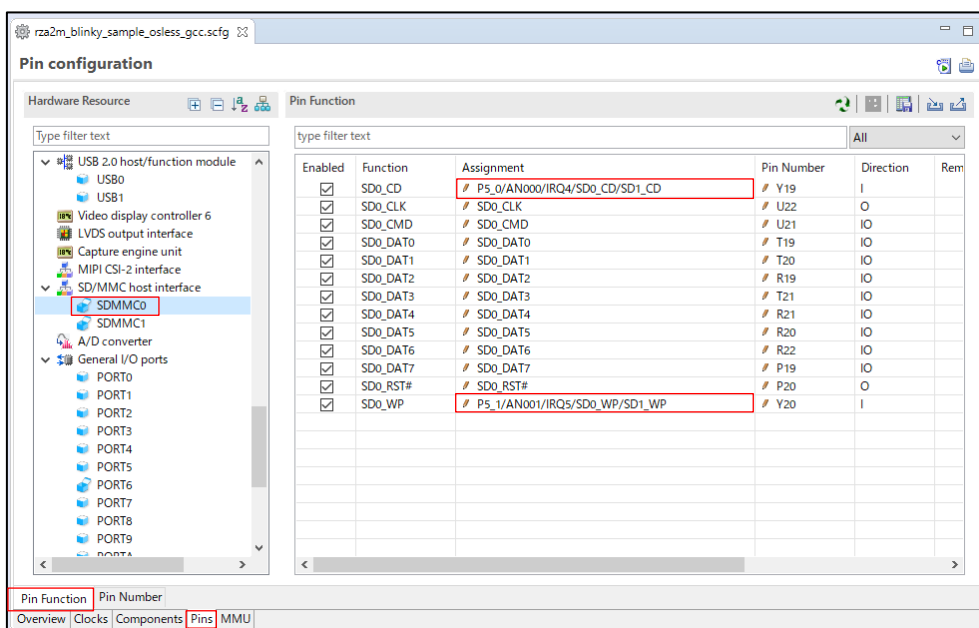


Figure 8.10 SD0_CD and SD0_WP pin assignments

8.3.2 PD_1 pin (SDVcc_SEL) settings

1. Select “General I/O ports”-“PORTD” from “Hardware Resource” and enable the PD_1 pin check box.

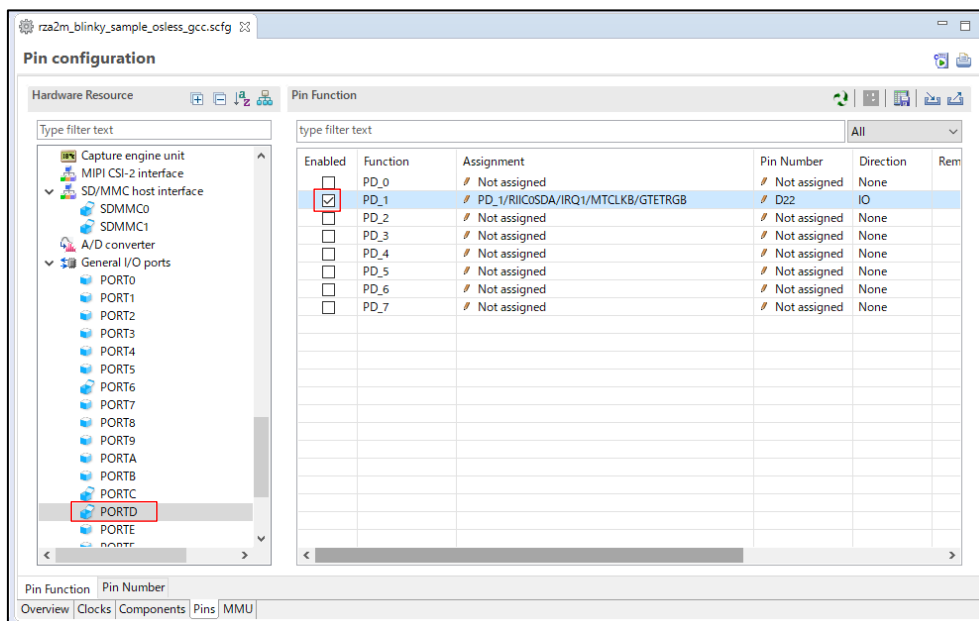


Figure 8.11 PD_1 pin selection

2. Select the “Pin Number” tab and set the PD_1 pin from the “Pin configuration”.

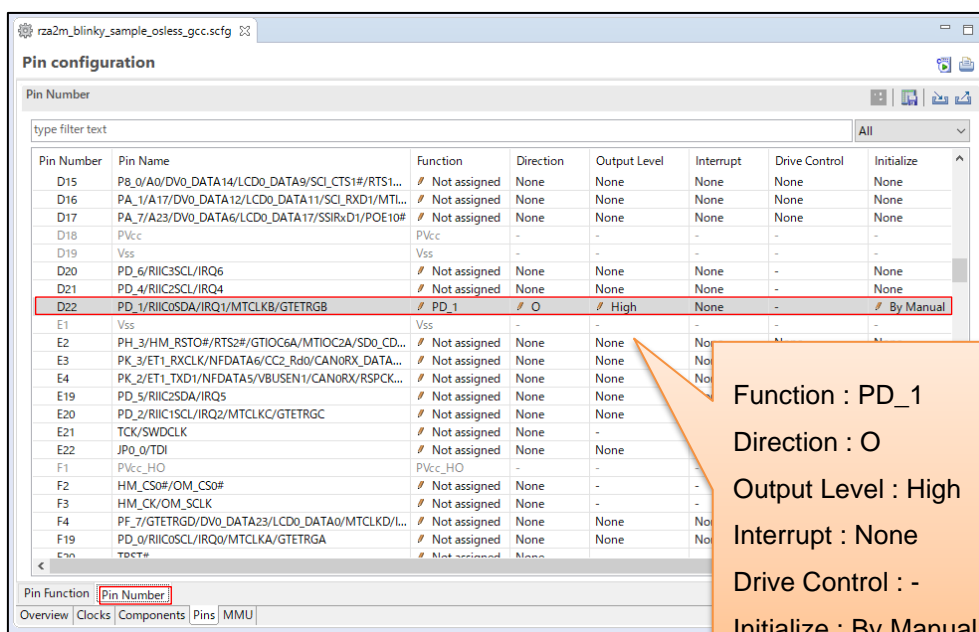


Figure 8.12 PD_1 pin settings

Function : PD_1
 Direction : O
 Output Level : High
 Interrupt : None
 Drive Control : -
 Initialize : By Manual

8.3.3 PJ_1 pin (SW3 key input) settings

1. Select "Pin Function" tab and enable the PJ_1 pin check box.

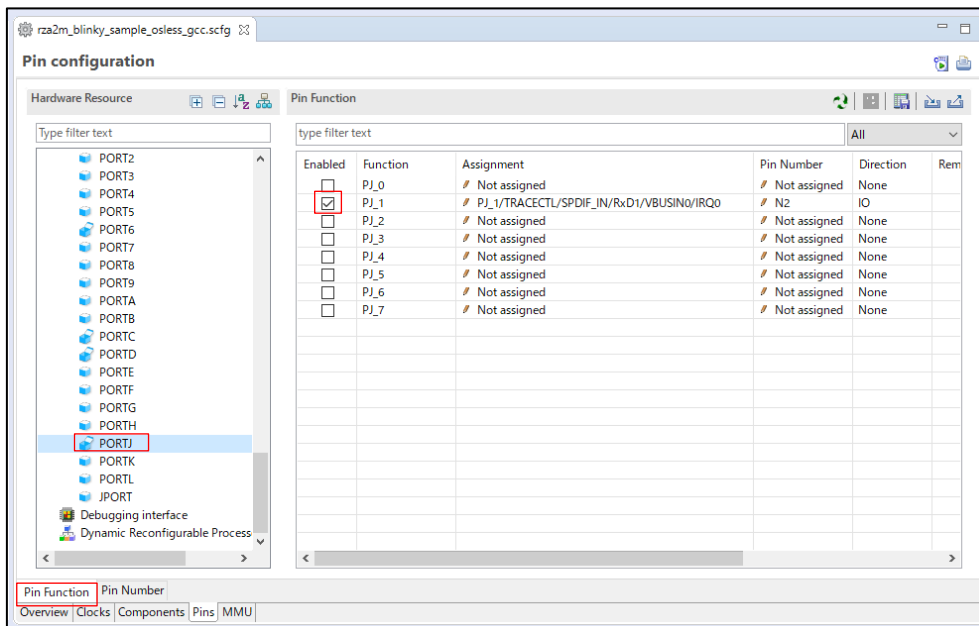


Figure 8.13 PJ_1 pin selection

2. Select "pin Number" tab and set PJ_1 pin.

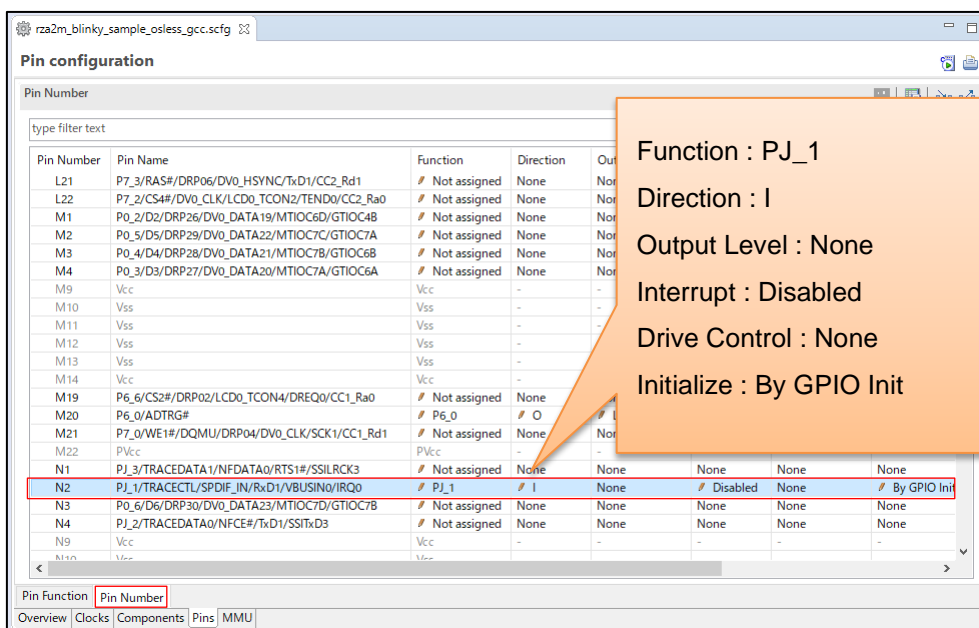


Figure 8.14 PJ_1 pin selection

8.3.4 PC_1 pin (LED1 (Yellowish-green)) settings

1. Select "Pin Function" tab and enable PC_1 pin check box.

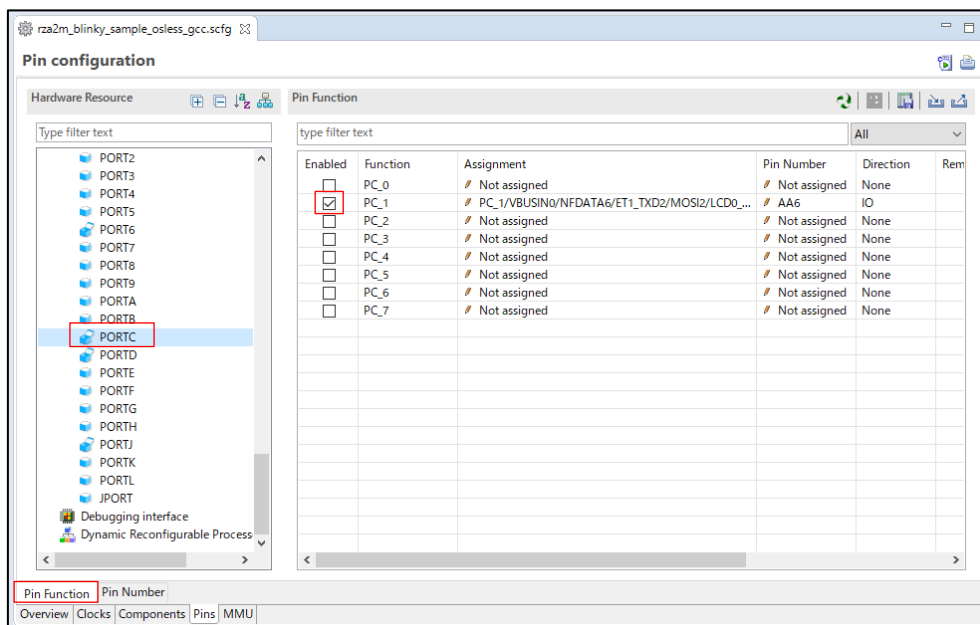


Figure 8.15 PC_1 pin selection

2. Select "Pin Number" tab and set PC_1 pin.

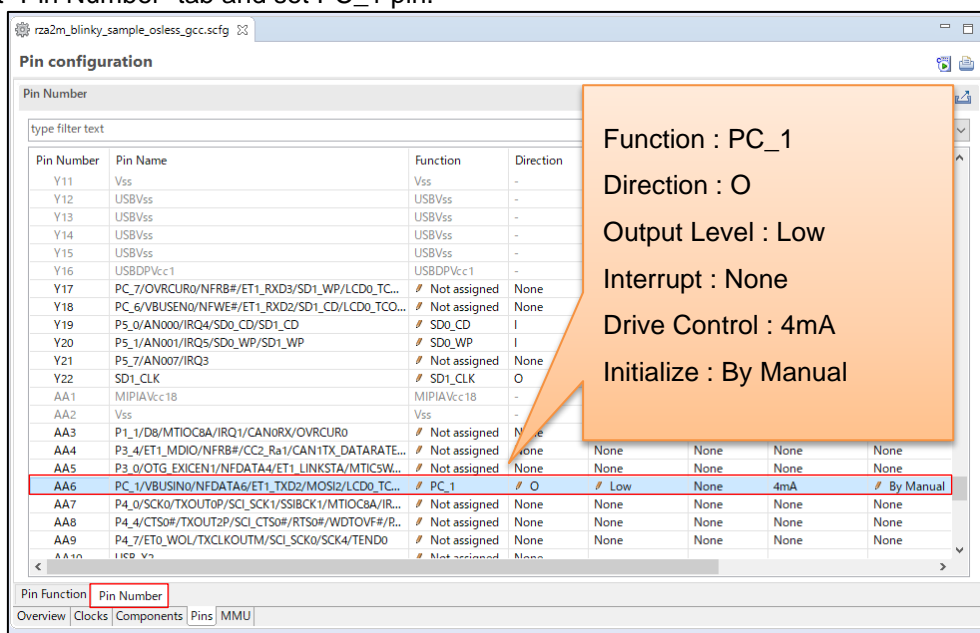


Figure 8.16 PC_1 pin settings

8.4 Code generation

The code generation procedure is described below.

1. Press the “Generate Code” button.

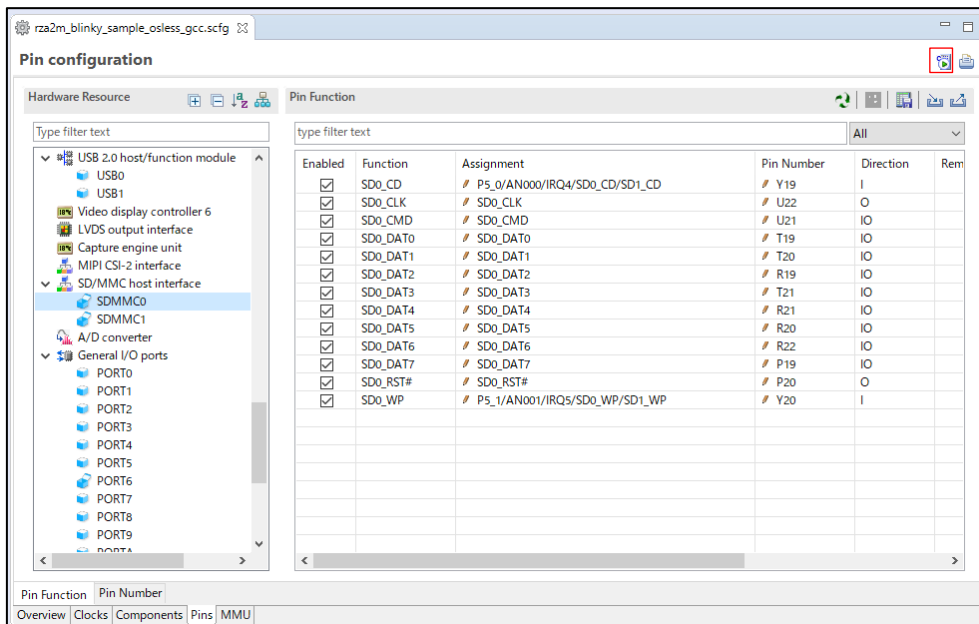


Figure 8.17 “Generate Code” selection

2. The code will be generated.

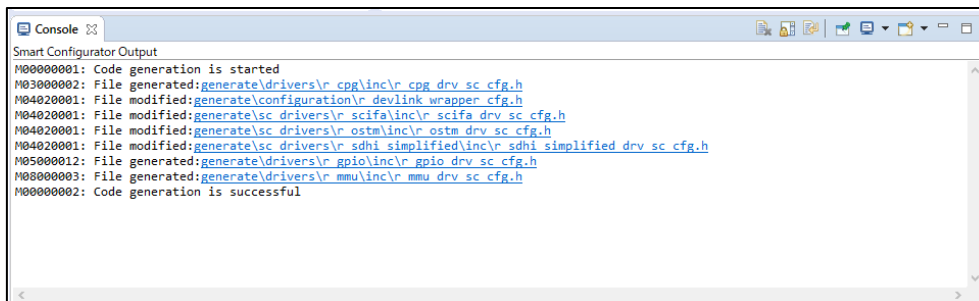


Figure 8.18 Console screen when generating code

9. Reference Documents

User's Manual: Hardware

RZ/A2M Group User's Manual: Hardware

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

RTK7921053C00000BE (RZ/A2M CPU board) User's Manual

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

RTK79210XXB00000BE (RZ/A2M SUB board) User's Manual

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

ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition Issue C

The latest version can be downloaded from the ARM website.

ARM Cortex™-A9 Technical Reference Manual Revision: r4p1

The latest version can be downloaded from the ARM website.

ARM Generic Interrupt Controller Architecture Specification - Architecture version 2.0

The latest version can be downloaded from the ARM website.

ARM CoreLink™ Level 2 Cache Controller L2C-310 Technical Reference Manual Revision: r3p3

The latest version can be downloaded from the ARM website.

Technical Update/Technical News

The latest information can be downloaded from the Renesas Electronics website.

User's Manual: Development Tools

Integrated development environment e2studio User's Manual can be downloaded from the Renesas Electronics website.

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

Specifications

SD Memory Card Specifications Part1 PHYSICAL LAYER Simplified SPECIFICATION, Ver6.00, August 29, 2018

Multi Media Card System Specifications, Ver. 4.1, Jan. 2005

Revision History

Rev.	Date	Description	
		Page	Summary
1.50	Mar.01.20	-	Added the SDIO function.
1.20	Jun.26.19	90	Table 5.1 Configuration Options
		104-113	Added SD card detection callback function setting. Added chapter 8, "Procedure to add component by Smart Configurator".
		94	Table 7.2 Operation Conformation Condition Remove item "Target".
		98	Table 7.5 SD Driver Memory Usage Updated the ROM size.
		-	The correction of the description.
1.10	May.17.19	94	Table 7.2 Operation Conformation Condition Remove compiler option "-mthumb-interwork"
1.00	Jan.01.19	-	First edition issued

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

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

2. Processing at power-on

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

3. Input of signal during power-off state

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

4. Handling of unused pins

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

5. Clock signals

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

6. Voltage application waveform at input pin

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

7. Prohibition of access to reserved addresses

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

8. Differences between products

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

Notice

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

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

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

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

6. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

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

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

(Rev.4.0-1 November 2017)

Corporate Headquarters

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

Trademarks

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

Contact information

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