RX Family

RSPI Module Using Firmware Integration Technology

R01AN1827EU0150
Rev. 1.50
Sep 30, 2016

Introduction

The RX family MCUs supported by this module have up to three channels of Serial Peripheral Interface (RSPI). RSPI is capable of high-speed, full-duplex synchronous serial communications with multiple processors and peripheral devices.

This document covers the RSPI Module Using Firmware Integration Technology (FIT) for the supported RX family MCUs. Details are provided that describe the RSPI driver's architecture, integration of the FIT module into a user's application, and how to use the API.

Target Device

The following is a list of devices that are currently supported by this API:

- **RX110 Group**
- **RX111 Group**
- **RX113 Group**
- **RX130 Group**
- **RX210 Group**
- **RX231, RX230 Group**
- **RX23T Group**
- **RX24T Group**
- **RX62N Group**
- **RX62T Group**
- **RX63N, RX631 Group**
- **RX64M Group**
- **RX65N Group**
- **RX71M Group**

When using this application note with other Renesas MCUs, careful evaluation is recommended after making modifications to comply with the alternate MCU.

Related Documents

- Firmware Integration Technology User's Manual (R01AN1833EU)
- Board Support Package Module Using Firmware Integration Technology (R01AN1685EU)
- Adding Firmware Integration Technology Modules to Projects (R01AN1723EU)
- Adding Firmware Integration Technology Modules to CubeSuite+ Projects (R01AN1826EJ)

Contents

1. Overview

This software provides an applications programing interface (API) to prepare the RSPI peripheral for operation and for performing data transfers over the SPI bus.

The RSPI Driver module fits between the user application and the physical hardware to take care of the low-level hardware control tasks that manage the RSPI peripheral.

It is recommended to review the RSPI peripheral chapter in the RX MCU hardware user's manual before using this software.

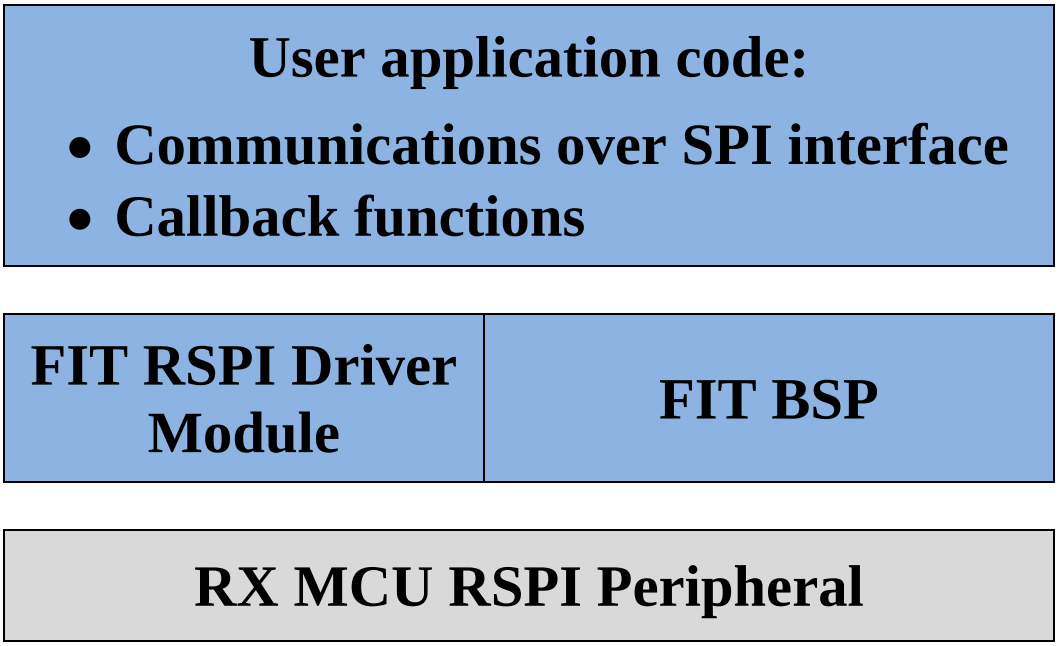

Figure 1 : Example Figure Showing Project Layers

1.1 Using the FIT RSPI Driver module

After adding the RSPI driver module to your project you will need to modify the *r_rspi_rx_config.h* file to configure the software for your installation. See Section 0 for details on configuration options.

Control registers for input and output signal pins to be used for RSPI must be correctly set up before the RSPI peripheral can operate with them. The RSPI module does not provide any means to initialize the pin registers that needs to be done externally prior to calling RSPI API functions. Typically this would be accomplished at system startup time as part of a general pin initialization routine.

When using an RSPI channel at run time, the first step is to call the R_RSPI_Open() function by passing the required settings and parameters. On completion, by setting up the I/O ports, the RSPI channel will be active and ready to perform all other functions available in this API. SPI Data transfer operations may be used at this time, or various control operations may be performed to change settings (Note 1).

Note 1: When using in clock synchronous operation (3-wire method) and in master mode, follow the procedure below to prepare for data transmission. Otherwise, clock may be asynchronized.
(1) Disable the slave for communication (For RSPI slave, set SPE=0)
(2) Call R_RSPI_Open()
(3) Set the pins to peripheral module by I/O ports setting
(4) Call R_RSPI_Write()
This processing is for dummy data transmission to determine the SPI mode.
(5) Enable the slave for communication

The setting of RSPI register other than RSPI command register (SPCMD) is applied by calling R_RSPI_Open().
As intended to general-purpose use, register's default value should be set in the RSPI register.
By calling R_RSPI_Control(), RSPI register information stored in RSPI driver module can be rewritten.

To reflect the rewritten information in the RSPI register, run the R_RSPI_Close(), then re-run R_RSPI_Open().

Three commands are provided in the R_RSPI_Control() function:

- Change the base bit-clock rate.
- Immediately abort a transfer operation.
- Rewrite RSPI register information stored in RSPI driver module.

When data transfers are performed over the SPI bus the driver informs the user's application of the completion status by calling the user-provided callback function.

Most of the RSPI API functions will require a 'handle' argument. This is used to identify the RSPI channel number that is selected for the operation. A handle is obtained by first calling the R_RSPI_Open() function. You must provide the address of a location where you will store the handle to R_RSPI_Open(), and on completion the handle will be available for use. Thereafter, simply pass the provided handle value for that RSPI channel number to the other API functions when calling them. In your application you will need to keep track of which handle belongs to a given channel, as each channel will be assigned its own handle.

2. API Information

This Driver API follows the Renesas API naming standards.

2.1 Hardware Requirements

This driver requires your MCU support the following features:

- One or more available RSPI peripheral channels.

2.2 Hardware Resource Requirements

This section details the hardware peripherals that this driver requires. Unless explicitly stated, these resources must be reserved for the driver and the user cannot use them independently.

- RSPI.

2.3 Software Requirements

This driver is dependent upon the support from the following software:

- This software depends on a FIT-compliant BSP module. The related I/O ports should be correctly initialized elsewhere after calling the R_RSPI_Open() of this software.
- This software requires that the peripheral clock (PCLKB) has been initialized by the BSP prior to calling the APIs of this module. The r_bsp macro ‘BSP_PCLKx_HZ’ is used by the driver for calculating bit-rate register settings. If the user modifies the PCLKx setting outside of the r_bsp module or the r_cgc module, then calculations on the bit rate will be invalid.

2.4 RSPI features Supported by Driver

This driver supports the following subset of the features available with the RSPI peripheral.

RSPI transfer functions:

- Use of MOSI (master out/slave in), MISO (master in/slave out), SSL (slave select), and RSPCK (RSPI clock) signals allows serial communications through SPI operation (four-wire method) or clock synchronous operation (three-wire method).
- Capable of serial communications in master/slave mode
- Switching of the polarity of the serial transfer clock
- Switching of the phase of the serial transfer clock

Data format:

- MSB-first/LSB-first selectable
- Transfer bit length is selectable as 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, or 32 bits.

Bit rate:

- In master mode, the on-chip baud rate generator generates RSPCK by frequency-dividing PCLK (Division ratio: 2 to 4096).
- In slave mode, the externally input clock is used as the serial clock (for maximum frequency, refer to MCU User's manual).

Error detection:

- Mode fault error detection

- Overrun error detection
- Parity error detection
- Under run detection

SSL control function:

- Four SSL signals (SSLn0 to SSLn3) for each channel
- In single-master mode: SSLn0 to SSLn3 signals are output.
- In slave mode: SSLn0 signal for input, selects the RSPI slave. SSLn1 to SSLn3 signals are unused.
- Controllable delay from SSL output assertion to RSPCK operation (RSPCK delay)

  Range: 1 to 8 RSPCK cycles (set in RSPCK-cycle units)
- Controllable delay from RSPCK stop to SSL output negation (SSL negation delay)

  Range: 1 to 8 RSPCK cycles (set in RSPCK-cycle units)
- Controllable wait for next-access SSL output assertion (next-access delay)

  Range: 1 to 8 RSPCK cycles (set in RSPCK-cycle + 2 PCLK units)
- Able to change SSL polarity

Control in master transfer:

- For each transfer operation, the following can be set:

  Slave select number, further division of base bit rate, SPI clock polarity/phase, transfer data bit-length, MSB/LSB-first, burst (holding SSL), SPI clock delay, slave select negation delay, and next-access delay

Interrupt sources:

- RSPI receive interrupt (receive buffer full)
- RSPI transmit interrupt (transmit buffer empty)
- RSPI error interrupt (mode fault, overrun, parity error, under run)

2.5 RSPI Features not supported

- Due to 16-bit or 32-bit data register access restrictions of the RSPI peripheral, DMAC/DTC support is somewhat complicated for other size data. DMAC/DTC transfers are not supported by this driver.
- To conserve limited RAM resources of smaller memory MCUs, like the RX111, this driver requires that data buffers are not statically allocated by the driver, but rather must be allocated by the user application at a higher level. This gives the application the control of how to allocate RAM.
- Only single-sequence data transfers are supported. The multi-command-sequence data transfer features of the RSPI peripheral are not supported by this driver.
- Only single-frame data transfers are supported by this driver. The multi-frame features of the RSPI peripheral are not supported. This means that the maximum supported data frame size is 32 bits.
- Multi-master mode is not supported.
- Byte swap for 16-bit type is not supported.

2.6 Supported Toolchains

This driver is tested and working with the following toolchains:

- Renesas RX Compiler Toolchain v2.01

2.7 Header Files

All API calls are accessed by including a single file "r_rspi_rx_if.h" which is supplied with this software's project code.

Build-time configuration options are selected or defined in the file "r_rspi_rx_config.h"

2.8 Integer Types

If your toolchain supports C99 then *stdint.h* should be described as shown below. If not, then there should be *typedefs.h* file that is included with your project as defined by the Renesas Coding Standards document.

This project uses ANSI C99 "Exact width integer types" in order to make the code clearer and more portable. These types are defined in *stdint.h*.

2.9 Code Size and RAM usage

The code size is based on optimization level 2 for the RXC toolchain. The ROM (code and constants) and permanently allocated RAM sizes vary based on the build-time configuration options set in the module configuration header file.

Sizes listed here are given for a minimum build configuration and a maximum build configuration. The minimum build includes one RSPI channel configured for use and all other optional features disabled. The maximum build sizes includes all available RSPI channels for the given MCU configured to be used, and the parameter checking and access locking options turned on. Stack usage is not listed, and should be determined by the user.

ROM and RAM Minimum and Maximum Sizes		
MCU max channels	**Minimum**	**Maximum**
RX111 1 channel	ROM: 1506 CODE + 0 DATA = 1506 bytes	ROM: 1753 CODE + 64 DATA = 1817 bytes
	RAM: 59 bytes	RAM: 63 bytes
RX231 1 channel	ROM: 1506 CODE + 0 DATA = 1506 bytes	ROM: 1751 CODE + 64 DATA = 1815 bytes
	RAM: 59 bytes	RAM: 63 bytes
RX64M 1 channel	ROM: 1585 CODE + 0 DATA = 1585 bytes	ROM: 1830 CODE + 64 DATA = 1894 bytes
	RAM: 59 bytes	RAM: 63 bytes
RX65N 3 channels	ROM: 1609 CODE + 0 DATA = 1609 bytes	ROM: 2494 CODE + 64 DATA = 2558 bytes
	RAM: 59 bytes	RAM: 189 bytes

Table 1: Minimum and Maximum ROM and RAM sizes

.

2.10 Configuration Overview

Some features or behavior of the software are determined at build-time by configuration options that the user must select.

Configuration options in *r_rspi_rx_config.h*	
RSPI_CFG_PARAM_CHECKING_ENABLE	Checking of arguments passed to RSPI API functions can be enabled or disabled. Disabling argument checking is provided for systems that absolutely require faster and smaller code. By default the module is configured to use the setting of the system-wide BSP_CFG_PARAM_CHECKING_ENABLE macro. This can be locally overridden for the RSPI module by redefining RSPI_CFG_PARAM_CHECKING_ENABLE. To control parameter checking locally, set RSPI_CFG_PARAM_CHECKING_ENABLE to 1 to enable it, otherwise set to 0 skip checking.
RSPI_CFG_REQUIRE_LOCK	If this is set to (1) then the RSPI driver will attempt to obtain the lock for the channel when performing certain operations to prevent concurrent access conflicts.
RSPI_CFG_DUMMY_TXDATA	The user-specified Dummy Data to be transmitted during receive-only operations.
RSPI_CFG_USE_CHANn	Enable the RSPI channels to use at build-time. (0) = not used. (1) = used.
RSPI_CFG_IR_PRIORITY_CHANn	Sets the shared interrupt priority for the channel. This is provided as a convenience. Priority can still be changed outside of this module at run time *after* a call to R_RSPI_Open has been made to a channel. However the next call to R_RSPI_Open for that channel will change it back to this configuration value.
RSPI_CFG_USE_RX63_ERROR_INTERRUPT	For RX63 group MCUs the RSPI error interrupt is a group interrupt shared with the SCI peripheral. So the error interrupt is disabled by default for RX63 group to prevent conflict with SCI FIT module. However, If not using the SCI FIT module, this may be enabled by setting RSPI_CFG_USE_RX63_ERROR_INTERRUPT to (1).
RSPI_CFG_MASK_UNUSED_BITS	When reading the RSPI received data register for data frame bit lengths that are not 8, 16, or 32-bits the unused upper bits will contain residual values from the transmit data. As a convenience, these unused upper bits may be optionally masked off (cleared to 0) by the driver when the data is transferred to the user-data buffer. Since this takes additional processing time in the data transfer interrupt handler it will cause a reduction in performance for the highest bit rate settings. This is not needed for 8, 16, or 32-bit transfers. So only enable this option when using data frame bit lengths that are not 8, 16, or 32-bits. (0) = do not clear. (1) = clear unused upper bits.

Table 2 : List of RSPI driver module configuration options

2.11 API Data Structures

This section details the data structures that are used with the driver's API functions.

2.11.1 Special Data Types

To provide strong type checking and reduce errors, many parameters used in API functions require arguments to be passed using the provided type definitions. Allowable values are defined in the public interface file *r_rspi_rx_if.h*. The following special types have been defined:

Enumeration of SPI bus interface modes

Type: rspi_interface_mode_t

Values: RSPI_IF_MODE_3WIRE // Use GPIO for slave select.

RSPI_IF_MODE_4WIRE // Use slave select signals controlled by RSPI.

Enumeration of master or slave operating mode configuration settings

Type: rspi_master_slave_mode_t

Values: RSPI_MS_MODE_MASTER // Channel operates as SPI master.

RSPI_MS_MODE_SLAVE // Channel operates as SPI slave

RSPI control command codes

Type: rspi_cmd_t

Values: RSPI_CMD_SET_BAUD // Use to set the base SPI clock bit-rate

RSPI_CMD_ABORT // Stop the current read or write operation immediately.

RSPI_CMD_SETREGS // Set additional RSPI regs in one operation. (Expert use only)

RSPI control command data structures

See R_RSPI_Control() chapter.

Handle

Type: rspi_handle_t

Values: User allocates storage for this type for a handle. The address of this location must be passed in the call to the R_RSPI_Open() function. The handle value is automatically assigned by R_RSPI_Open() function and returned in the location specified.

Channel Settings structure for Open

The R_RSPI_Open() function requires a pointer to an initialized instance of this structure to set certain operating modes at the channel open.

Type: rspi_chnl_settings_t

Members: rspi_interface_mode_t gpio_ssl; // Specify the interface mode.

rspi_master_slave_mode_t master_slave_mode; // Specify master or slave mode operation.

uint32_t bps_target; // The target bits per second setting for the channel.

Callback function data structure

The channel number and the procedure result code are passed in this data structure to the user defined callback function.

Type: rspi_callback_data_t

Members: rspi_handle_t handle; // The channel handle.

rspi_evt_t event_code; // The event code.

2.12 Typedef enumerations used for the command settings word

This list contains the enumerated types available for specific settings of the command word for write and read operations. The command word is a 32-bit value that is a collection of bit fields. Note that the valid data is lower 16 bits. The lower 16-bit data will get copied to the SPCMD register for each call to one of the read or write functions. To build a complete command lower 16-bit data select one and only one member from each type and assign it to the corresponding member in the rspi_command_word_t structure. For lower 16 bits, set the dummy data (RSPI_SPCMD_DUMMY).

Clock phase

The combination of the CPHA (clock phase) and CPOL (clock resting polarity) determine the "SPI mode setting"

Note: For slave-mode operation RSPI only supports sampling on even edge. This corresponds to what is sometimes referred to as SPI Mode-1, or Mode-3.

Type: rspi_spcmd_cpha_t

Members: RSPI_SPCMD_CPHA_SAMPLE_ODD // Data sampling on odd edge, data variation on even edge.
RSPI_SPCMD_CPHA_SAMPLE_EVEN // Data variation on odd edge, data sampling on even edge.

Clock polarity

Type: rspi_spcmd_cpol_t

Members: RSPI_SPCMD_CPOL_IDLE_LO // RSPCK is low when idle.
RSPI_SPCMD_CPOL_IDLE_HI // RSPCK is high when idle.

Clock base rate division

The SPI clock base bit rate setting will be further divided by this. (Note 1)

Type: rspi_spcmd_br_div_t

Members: RSPI_SPCMD_BR_DIV_1 // Select the base bit rate
RSPI_SPCMD_BR_DIV_2 // Select the base bit rate divided by 2
RSPI_SPCMD_BR_DIV_4 // Select the base bit rate divided by 4
RSPI_SPCMD_BR_DIV_8 // Select the base bit rate divided by 8

Note 1 : The bit rate specified in R_RSPI_Open() or R_RSPI_Control() is based on no frequency division (RSPI_SPCMD_BR_DIV_1). To divide the selected bit rate, change the setting of this bit.

Slave select to be asserted during transfer operation.

Type: rspi_spcmd_ssl_assert_t

Members: RSPI_SPCMD_ASSERT_SSL0 // Select SSL0
RSPI_SPCMD_ASSERT_SSL1 // Select SSL1
RSPI_SPCMD_ASSERT_SSL2 // Select SSL2
RSPI_SPCMD_ASSERT_SSL3 // Select SSL3

Slave select negation.

This bit determines whether the RSPI will deassert the slave select signal after each frame, or keep it asserted.

Type: rspi_spcmd_ssl_negation_t

Members: RSPI_SPCMD_SSL_NEGATE // Negates all SSL signals upon completion of transfer.
RSPI_SSPCMD_SL_KEEP // Keep SSL signal level from end of transfer until start of next.

Frame data length

The number of bits in each SPI data frame.

Type: rspi_spcmd_bit_length_t

Members: RSPI_SPCMD_BIT_LENGTH_8 // 8 bits data length
RSPI_SPCMD_BIT_LENGTH_9 // 9 bits data length
RSPI_SPCMD_BIT_LENGTH_10 // 10 bits data length
RSPI_SPCMD_BIT_LENGTH_11 // 11 bits data length
RSPI_SPCMD_BIT_LENGTH_12 // 12 bits data length
RSPI_SPCMD_BIT_LENGTH_13 // 13 bits data length
RSPI_SPCMD_BIT_LENGTH_14 // 14 bits data length
RSPI_SPCMD_BIT_LENGTH_15 // 15 bits data length

	RSPI_SPCMD_BIT_LENGTH_16	// 16 bits data length
	RSPI_SPCMD_BIT_LENGTH_20	// 20 bits data length
	RSPI_SPCMD_BIT_LENGTH_24	// 24 bits data length
	RSPI_SPCMD_BIT_LENGTH_32	// 32 bits data length

Data transfer bit order.

Type: rspi_spcmd_bit_order_t

Members:	RSPI_SPCMD_ORDER_MSB_FIRST	// MSB first.
	RSPI_SPCMD_ORDER_LSB_FIRST	// LSB first.

RSPI signal delays

Type: rspi_spcmd_spnden_t

Members:	RSPI_SPCMD_NEXT_DLY_1	// A next-access delay of 1 RSPCK +2 PCLK.
	RSPI_SPCMD_NEXT_DLY_SSLND	// Next-access delay = next access delay register (SPND)

Type: rspi_spcmd_slnden_t

Members:	RSPI_SPCMD_SSL_NEG_DLY_1	// An SSL negation delay of 1 RSPCK.
	RSPI_SPCMD_SSL_NEG_DLY_SSLND	// Delay = SSL negation delay register (SSLND)

Type: rspi_spcmd_sckden_t

Members:	RSPI_SPCMD_CLK_DLY_1	// An RSPCK delay of 1 RSPCK.
	RSPI_SPCMD_CLK_DLY_SPCKD	// Delay = setting of RSPI clock delay register (SPCKD).

Dummy data

Type: rspi_spcmd_dummy_t

Members:	RSPI_SPCMD_DUMMY	upper 16-bit dummy data

2.12.1 Complete command word data structure.

This contains one of each of the above types in the correct order to set all the bits of the SPCMD register.

```
#pragma bit_order right      // Match the order of description in the hardware manual.
typedef union rspi_command_word_s
{
    struct{
        rspi_spcmd_cpha_t           cpha            :1;
        rspi_spcmd_cpol_t           cpol            :1;
        rspi_spcmd_br_div_t         br_div          :2;
        rspi_spcmd_ssl_assert_t     ssl_assert      :3;
        rspi_spcmd_ssl_negation_t   ssl_negate      :1;
        rspi_spcmd_bit_length_t     bit_length      :4;
        rspi_spcmd_bit_order_t      bit_order       :1;
        rspi_spcmd_spnden_t         next_delay      :1;
        rspi_spcmd_slnden_t         ssl_neg_delay   :1;
        rspi_spcmd_sckden_t         clock_delay     :1;
        rspi_spcmd_dummy_t          dummy           :16
    };
    uint16_t word[2];
} rspi_command_word_t;
```

Example of command word initialization

```
static const rspi_command_word_t my_command_reg_word = {
        RSPI_SPCMD_CPHA_SAMPLE_ODD,
        RSPI_SPCMD_CPOL_IDLE_LO,
        RSPI_SPCMD_BR_DIV_1,
        RSPI_SPCMD_ASSERT_SSL0,
        RSPI_SPCMD_SSL_KEEP,
        RSPI_SPCMD_BIT_LENGTH_8,
        RSPI_SPCMD_ORDER_MSB_FIRST,
        RSPI_SPCMD_NEXT_DLY_SSLND,
        RSPI_SPCMD_SSL_NEG_DLY_SSLND,
        RSPI_SPCMD_CLK_DLY_SPCKD,
        RSPI_SPCMD_DUMMY,
};
```

2.13 Return Values

The different values API functions can return.

Return Type: rspi_err_t

Values:	Cause
RSPI_SUCCESS	Function completed without errors
RSPI_ERR_BAD_CHAN	Invalid channel number
RSPI_ERR_CH_NOT_OPENED	Channel not yet opened. Function cannot be completed.
RSPI_ERR_CH_NOT_CLOSED	Channel still open from previous open.
RSPI_ERR_UNKNOWN_CMD	Control command is not recognized.
RSPI_ERR_INVALID_ARG	Argument is not valid for parameter.
RSPI_ERR_ARG_RANGE	Argument is out of range for parameter.
RSPI_ERR_NULL_PTR	Received null pointer; missing required argument.
RSPI_ERR_LOCK	A lock procedure failed
RSPI_ERR_UNDEF	Undefined/unknown error

2.14 Event Codes

The different codes returned by API events.

Return Type: rspi_evt_t

Values:	Cause
RSPI_EVT_TRANSFER_COMPLETE	The data transfer completed.
RSPI_EVT_TRANSFER_ABORTED	The data transfer was aborted.
RSPI_EVT_ERR_MODE_FAULT	Mode fault error.
RSPI_EVT_ERR_READ_OVF	Read overflow.
RSPI_EVT_ERR_PARITY	Parity error.
RSPI_EVT_ERR_UNDER_RUN	Underrun error
RSPI_EVT_ERR_UNDEF	Undefined/unknown error event.

2.15 Adding the RSPI Driver FIT Module to Your Project

This module must be added to each project in the e^2 Studio.

There are two methods for adding to a project: using the FIT plug-in and adding manually.

When the FIT plug-in is used, FIT modules can be added to projects easily and the include file path will be updated automatically. Therefore we recommend using the FIT plug-in when adding FIT modules to a project.

There are the following methods to add FIT module using FIT plug in.

1. Use "FIT Configurator".
   This is the latest method that the plug-in function such as Lib file path automatic setting is enhanced, which we recommend the use.

   For the procedure, refer to “4.3.2 Install the FIT Modules with the FIT Plugin.” in “RX64M/RX71M Group RX Driver Package Ver.1.02 (R01AN2606EJ)” application note.

2. Use the existing ”FIT plug-in”.
   For the procedure, refer to “3. Adding FIT Modules to e2 studio Projects FIT Plug-In” in “Adding Firmware Integration Technology Modules to Projects (R01AN1723EU)” application note.

3. API Functions

3.1 Summary

The following functions are included in this design:

Function	Description
R_RSPI_Open()	Initializes the associated registers required to prepare the specified RSPI channel for use, provides the handle for use with other API functions. Takes a callback function pointer for responding to interrupt events.
R_RSPI_Close()	Disables the specified RSPI channel.
R_RSPI_Control()	Handles special hardware or software operations for the RSPI channel.
R_RSPI_Write()	The Write function transmits data to a SPI master or slave device.
R_RSPI_Read()	The Read function receives data from a SPI master or slave device.
R_RSPI_WriteRead()	The Write Read function simultaneously transmits data to a SPI master or slave device while receiving data from that device (full duplex).
R_RSPI_GetVersion()	Returns the driver version number.

Table 3: List of RSPI API functions

3.2 R_RSPI_Open()

This function applies power to the RSPI channel, initializes the associated registers, enables interrupts, and provides the channel handle for use with other API functions.

Format

```
rspi_err_t   R_RSPI_Open(uint8_t                 channel,
                         rspi_chnl_settings_t  *pconfig,
                         void                 (*pcallback)(void *pcbdat),
                         rspi_handle_t         *phandle);
```

Parameters

channel

Number of the RSPI channel to be initialized

*pconfig

Pointer to RSPI channel configuration data structure.

(*pcallback)(void * pcbdat)

Pointer to user defined function called from interrupt.

*phandle

Pointer to a handle for channel. Handle value will be set by this function

Return Values

RSPI_SUCCESS:	Successful; channel initialized
RSPI_ERR_BAD_CHAN:	Channel number is not available
RSPI_ERR_CH_NOT_CLOSED:	Channel currently in operation; Perform R_RSPI_Close() first
RSPI_ERR_NULL_PTR:	*pconfig pointer or *phandle pointer is NULL
RSPI_ERR_INVALID_ARG:	An element of the *pconfig structure contains an invalid value.
RSPI_ERR_LOCK:	The lock could not be acquired.

Properties

Prototyped in file “r_rspi_rx_if.h”

Description

The Open function is responsible for preparing an RSPI channel for operation. This function must be called once prior to calling any other RSPI API functions (except R_RSPI_GetVersion). Once successfully completed, the status of the selected RSPI will be set to "open". After that, this function should not be called again for the same RSPI channel without first performing a "close" by calling R_RSPI_Close().

Communication is not yet available upon completion of this processing. Set MPC and PMR in the I/O ports to peripheral module.

Reentrant

Yes. Reentrance for different channel OK. Reentrance for the same channel will result in a ‘Lock error’ return.

Example

```
/* Conditions: Channel not yet open. */
uint8_t chan = 0;
rspi_handle_t handle;
rspi_chnl_settings_t my_config;
rspi_cmd_baud_t my_setbaud_struct;
rspi_err_t rspi_result;

my_config.gpio_ssl          = RSPI_IF_MODE_4WIRE;
my_config.master_slave_mode = RSPI_MS_MODE_MASTER;
my_config.bps_target        = 4000000; // Bit rate in bits-per-second.

rspi_result = R_RSPI_Open(chan, &my_config, &test_callback, &handle );

if (RSPI_SUCCESS != rspi_result)
{
    return rspi_result;
}

/* Initialize I/O port pins for use with the RSPI peripheral.
 * This is specific to the MCU and ports chosen. */
 rspi_64M_init_ports();
```

3.3 R_RSPI_Control()

The Control function is responsible for handling special hardware or software operations for the RSPI channel.

Format

```
rspi_err_t     R_RSPI_Control(rspi_handle_t  handle,
                              rspi_cmd_t     cmd,
                              void          *pcmd_data);
```

Parameters

handle

Handle for the channel

cmd

Enumerated command code.

Available command codes:

RSPI_CMD_SET_BAUD // Change the base bit rate setting without reinitializing the RSPI channel.
RSPI_CMD_ABORT, // Stop the current read or write operation immediately.
RSPI_CMD_SETREGS, // Set all supported RSPI regs in one operation. (Expert use only)

*pcmd_data

Pointer to the command-data structure parameter of type void that is used to reference the location of any data specific to the command needed for its completion. Commands that do not require supporting data must use the FIT_NO_PTR

Return Values

RSPI_SUCCESS:	Command successfully completed.
RSPI_ERR_CH_NOT_OPEN:	The channel has not been opened. Perform R_RSPI_Open() first
RSPI_ERR_BAD_CHAN:	Channel number is not available.
RSPI_ERR_UNKNOWN_CMD:	Control command is not recognized.
RSPI_ERR_NULL_PTR:	*pcmd_data pointer or *phandle pointer is NULL.
RSPI_ERR_INVALID_ARG:	An element of the *pcmd_data structure contains an invalid value.
RSPI_ERR_LOCK:	The lock could not be acquired.

Properties

Prototyped in file “r_rspi_rx_if.h”

Description

This function is responsible for handling special hardware or software operations for the RSPI channel. It takes an RSPI handle to identify the selected RSPI, an enumerated command value to select the operation to be performed, and a void pointer to a location that contains information or data required to complete the operation. This pointer must point to storage that has been type-cast by the caller for the particular command using the appropriate type provided in "r_rspi_rx_if.h".

Command	argument pcmd_data	Description
RSPI_CMD_SET_BAUD	rspi_cmd_baud_t *	Change the bit rate setting without reinitializing the RSPI channel.
RSPI_CMD_ABORT	FIT_NO_PTR	Stop the current read or write operation immediately.
RSPI_CMD_SETREGS	rspi_cmd_setregs_t *	Set all supported RSPI regs in one operation. Expert use only.

Reentrant

Other than the ABORT command, which is reentrant safe, the following applies:

Reentrancy to operate on the same RSPI is rejected by BSP lock if locking is enabled.

If locking is disabled then reentrancy is only safe for operating on a different RSPI. If locking is enabled special care should be taken to prevent deadlock. Caller should check return value and should handle a locked return status by permitting the process owning the lock to complete.

Example

```
my_setbaud_struct.bps_target = 4000000; // Set for 4 Mbps
rspi_result = R_RSPI_Control(handle, RSPI_CMD_SET_BAUD,  &my_setbaud_struct);
if (RSPI_SUCCESS != rspi_result)
{
    return error;
}

...
/* This is taking too long, stop the current transfer now! */
rspi_result = R_RSPI_Control(handle, RSPI_CMD_ABORT,  FIT_NO_PTR);
```

Type defines used with the R_RSPI_Control function.

Control function command codes.

```
typedef enum rspi_cmd_e
{
    RSPI_CMD_SET_BAUD = 1,
    RSPI_CMD_ABORT,     // Stop the current read or write operation immediately.
    RSPI_CMD_SETREGS,   // Set all supported RSPI regs in one operation.
} rspi_cmd_t;
```

Data structure for the Set Baud command. This command sets the base-bit rate for the specified channel. The value specified in ‘bps_target’ may not be what actually gets set. The function will try to find a setting to match, but if the requested bit rate is not possible based on the divisor ratios available, then the function will set the next lower available bit-rate. SPCMD.BRDV[1:0] bit is based on zero (no frequency division).

```
typedef struct rspi_cmd_baud_s
{
    uint32_t    bps_target;   // The target bits-per-second setting for the channel.
} rspi_cmd_baud_t;
```

Expert use only. Using the RSPI_CMD_SETREGS command, multiple RSPI registers setting information held in the driver can be changed. To use RSPI_CMD_SETREGS command, create the instance with as-needed setting value first, then call R_RSPI_Control() to pass the pointer as argument.

Note that the value of the RSPI register is unchanged at the time of running RSPI_CMD_SETREGS command. After this processing, the value of the RSPI register will be changed for the first time by calling R_RSPI_Close() and R_RSPI_Open().

```
typedef struct rspi_cmd_setregs_s
{
    uint8_t sslp_val;   /* RSPI Slave Select Polarity Register (SSLP) */
    uint8_t sppcr_val;  /* RSPI Pin Control Register (SPPCR) */
    uint8_t spckd_val;  /* RSPI Clock Delay Register (SPCKD) */
    uint8_t sslnd_val;  /* RSPI Slave Select Negation Delay Register (SSLND) */
    uint8_t spnd_val;   /* RSPI Next-Access Delay Register (SPND) */
    uint8_t spcr2_val;  /* RSPI Control Register 2 (SPCR2) */
    uint8_t spdcr2_val; /* RSPI Data Control Register 2 (SPDCR2) */
} rspi_cmd_setregs_t;
```

Special Notes:

The RSPI_CMD_ABORT command requires no supporting data, so `FIT_NO_PTR` is used as the pcmd_data argument.

3.4 R_RSPI_Close()

Fully disables the RSPI channel designated by the handle.

Format

```
RSPI_err_t      R_RSPI_Close(rspi_handle_t handle);
```

Parameters

handle

Handle for the channel

Return Values

RSPI_SUCCESS:	Successful; channel closed
RSPI_ERR_CH_NOT_OPEN:	The channel has not been opened so closing has no effect.
RSPI_ERR_BAD_CHAN:	Channel number is not available
RSPI_ERR_NULL_PTR:	A required pointer argument is NULL

Properties

Prototyped in file “r_rspi_rx_if.h”

Description

This disables the RSPI channel designated by the handle. The RSPI handle is modified to indicate that it is no longer in the 'open' state. The RSPI channel cannot be used again until it has been reopened with the R_RSPI_Open function. If this function is called for an RSPI that is not in the open state then an error code is returned.

Reentrant

Yes, however unintentional reentrancy for same RSPI channel may result in an unexpected RSPI_ERR_ NOT_OPEN return code.

Example

```
RSPI_err_t  rspi_result;

rspi_result = R_RSPI_Close(handle);

if (RSPI_SUCCESS != rspi_result)
{
    return rspi_result;
}
```

3.5 R_RSPI_Write()

The Write function transmits data to the selected SPI device

Format

```
rspi_err_t     R_RSPI_Write(rspi_handle_t       handle,
                            rspi_command_word_t spcmd_command_word,
                            void               *psrc,
                            uint16_t            length);
```

Parameters

handle

Handle for the channel

spcmd_command_word

Bit field data consisting of all the RSPI command register settings for SPCMD for this operation.
See: 2.12 Typedef enumerations used for the command settings word.

*psrc

Void type pointer to a source data buffer from which data will be transmitted to a SPI device. Based on the data frame bit-length specified in the spcmd_command_word.bit_length, the *psrc pointer will be type cast to the corresponding data type during the transfer. So, for example, if the bit-length is set to 16-bits, then the source buffer data will be accessed as a block of 16-bit data, and so on for each bit-length setting. Bit-length settings that are not 8, 16 or 32 will use the data type that they can be contained within. For example, 24-bit frames will be stored in 32-bit storage, 11-bit frames will be stored in 16-bit storage, etc.

length

Transfer length variable to indicate the number of data frames to be transferred. The size of the data word is determined from settings in the spcmd_command_word.bit_length argument. Be sure that the length argument matches the storage type of the source data; this is a count of the number of frames, not the number of bytes.

Return Values

RSPI_SUCCESS:	Write operation successfully completed.
RSPI_ERR_CH_NOT_OPEN:	The channel has not been opened. Perform R_RSPI_Open() first.
RSPI_ERR_BAD_CHAN:	Channel number is not available.
RSPI_ERR_NULL_PTR:	A required pointer argument is NULL.
RSPI_ERR_LOCK:	The lock could not be acquired. The channel is busy.

Properties

Prototyped in file “r_rspi_rx_if.h”

Description

Starts transmission of data to a SPI device. The function returns immediately after the transmit operation begins, and data will continue to be transmitted in the background under interrupt control until the requested length has been transmitted. When the transmission is complete the user-defined callback function is called. The callback function should be used to notify the user application that the transfer has completed.

Operation differs slightly depending on whether the RSPI is operating as Master or Slave. If the RSPI is configured as slave, then data will only transfer when clocks are received from the Master. Data received by the RSPI peripheral will be discarded.

Reentrant

Yes. Reentrance for different channel OK. Reentrance for the same channel will result in a ‘Lock error’ return.

Example

```
/* Conditions: Channel currently open. */
g_transfer_complete = false;

rspi_result = R_RSPI_Write(handle, my_command_word, source, length);
if (RSPI_SUCCESS != rspi_result)
{
   if (RSPI_ERR_LOCK == rspi_result)
   {
       // Channel must be busy. Try again later.
   }
   return error;
}

while (!g_transfer_complete) // Poll for interrupt callback to set this.
{
    nop(); // Do something useful while waiting for the transfer to complete.
}
```

3.6 R_RSPI_Read()

The Read function receives data from the selected SPI device.

Format

```
rspi_err_t     R_RSPI_Read(rspi_handle_t         handle,
                           rspi_command_word_t   spcmd_command_word,
                           void                 *pdest,
                           uint16_t              length);
```

Parameters

handle

Handle for the channel

spcmd_command_word

Bit field data consisting of all the RSPI command register settings for SPCMD for this operation.
See: 2.12 Typedef enumerations used for the command settings word.

pdest

Void type pointer to a destination buffer into which data will be copied that has been received from a SPI device. It is the responsibility of the caller to ensure that adequate space is available to hold the requested data count. The argument must not be NULL. Based on the data frame bit-length specified in the spcmd_command_word.bit_length, the *pdest pointer will be type cast to the corresponding data type during the transfer. So, for example, if the bit-length is set to 16-bits, then the data will be stored in the destination buffer as a 16-bit value, and so on for each bit-length setting. Bit-length settings that are not 8, 16 or 32 will use the smallest data type that they can be contained within. For example, 24-bit frames will be stored in 32-bit storage, 11-bit frames will be stored in 16-bit storage, etc.

length

Transfer length variable to indicate the number of data frames to be transferred. The size of the data word is determined from settings in the spcmd_command_word.bit_length argument. Be sure that the length argument matches the storage type of the source data; this is a count of the number of frames, not the number of bytes.

Return Values

RSPI_SUCCESS:	Read operation successfully completed.
RSPI_ERR_CH_NOT_OPEN:	The channel has not been opened. Perform R_RSPI_Open() first.
RSPI_ERR_BAD_CHAN:	Channel number is not available.
RSPI_ERR_NULL_PTR:	A required pointer argument is NULL.
RSPI_ERR_LOCK:	The lock could not be acquired. The channel is busy.

Properties

Prototyped in file “r_rspi_rx_if.h”

Description

Starts reception of data from a SPI device. The function returns immediately after the operation begins, and data will continue to be received in the background under interrupt control until the requested length has been received. Received data is stored in the destination buffer. When the transfer is complete the user-defined callback function is called.

Operation differs slightly depending on whether the RSPI is operating as Master or Slave. If the RSPI is configured as slave, then data will only transfer when clocks are received from the Master. While receiving data, the RSPI will also transmit the user definable Dummy data pattern defined in the configuration file.

Reentrant

Yes. Reentrance for different channel OK. Reentrance for the same channel will result in a ‘Lock error’ return.

Example

```
/* Conditions: Channel currently open. */
...
g_transfer_complete = false; // state flag variable defined elsewhere.

rspi_result = R_RSPI_Read(handle, my_command_word, dest, length);
if (RSPI_SUCCESS != rspi_result)
{
   return error;
}

while (!g_transfer_complete) // Poll for interrupt callback to set this.
{
    nop(); // Do something useful while waiting for the transfer to complete.
}
```

3.7 R_RSPI_WriteRead()

The Write Read function simultaneously transmits data to a SPI device while receiving data from a SPI device.

Format

```
rspi_err_t     R_RSPI_WriteRead(rspi_handle_t         handle,
                                rspi_command_word_t   spcmd_command_word,
                                void                 *psrc,
                                void                 *pdest,
                                uint16_t              length);
```

Parameters

handle

Handle for the channel

spcmd_command_word

Bit field data consisting of all the RSPI settings for the command register (SPCMD) for this operation. See: 2.12 Typedef enumerations used for the command settings word.

*psrc

Void type pointer to a source data buffer from which data will be transmitted to a SPI device. Based on the data frame bit-length specified in the spcmd_command_word.bit_length, the *psrc pointer will be type cast to the corresponding data type during the transfer. So, for example, if the bit-length is set to 16-bits, then the source buffer data will be accessed as a block of 16-bit data, and so on for each bit-length setting. Bit-length settings that are not 8, 16 or 32 will use the data type that they can be contained within. For example, 24-bit frames will be stored in 32-bit storage, 11-bit frames will be stored in 16-bit storage, etc.

*pdest

Void type pointer to a destination buffer into which data will be copied that has been received from a SPI device. It is the responsibility of the caller to ensure that adequate space is available to hold the requested data count. The argument must not be NULL. Based on the data frame bit-length specified in the spcmd_command_word.bit_length, the *pdest pointer will be type cast to the corresponding data type during the transfer. So, for example, if the bit-length is set to 16-bits, then the data will be stored in the destination buffer as a 16-bit value, and so on for each bit-length setting. Bit-length settings that are not 8, 16 or 32 will use the smallest data type that they can be contained within. For example, 24-bit frames will be stored in 32-bit storage, 11-bit frames will be stored in 16-bit storage, etc.

length

Transfer length variable to indicate the number of data frames to be transferred. The size of the data word is determined from settings in the spcmd_command_word.bit_length argument. Be sure that the length argument matches the storage type of the source data; this is a count of the number of frames, not the number of bytes. The same number of frames will be both written and read.

Return Values

RSPI_SUCCESS:	Read operation successfully completed.
RSPI_ERR_CH_NOT_OPEN:	The channel has not been opened. Perform R_RSPI_Open() first.
RSPI_ERR_BAD_CHAN:	Channel number is not available.
RSPI_ERR_NULL_PTR:	A required pointer argument is NULL.
RSPI_ERR_LOCK:	The lock could not be acquired. The channel is busy.

Properties

Prototyped in file “r_rspi_rx_if.h”

Description

Starts full-duplex transmission and reception of data to and from a SPI device. The function returns immediately after the transfer operation begins, and data transfer will continue in the background under interrupt control until the requested length has been transferred. When the operation is complete the user-defined callback function is called. The callback function should be used to notify the user application that the transfer has completed.

Operation differs slightly depending on whether the RSPI is operating as Master or Slave. If the RSPI is configured as slave, then data will only transfer when clocks are received from the Master. Data to be transmitted is obtained from the source buffer, while received data is stored in the destination buffer.

Reentrant

Yes. Reentrance for different channel OK. Reentrance for the same channel will result in a ‘Lock error’ return.

Example

```
/* Conditions: Channel currently open. */
g_transfer_complete = false;
rspi_result = R_RSPI_WriteRead(handle, my_command_word, source, dest, length);
if (RSPI_SUCCESS != rspi_result)
{
    return error;
}

while (!g_transfer_complete) // Poll for interrupt callback to set this.
{
    nop();// Do something useful while waiting for the transfer to complete.
}
```

3.8 R_RSPI_GetVersion()

This function returns the driver version number at runtime.

Format

```
uint32_t  R_RSPI_GetVersion(void);
```

Parameters

None

Return Values

Version number with major and minor version digits packed into a single 32-bit value.

Properties

Prototyped in file “r_rspi_rx_if.h”

Description

The function returns the version of this module. The version number is encoded such that the top two bytes are the major version number and the bottom two bytes are the minor version number.

Reentrant

Yes

Example

```
/* Retrieve the version number and convert it to a string. */

uint32_t   version, version_high, version_low;
char       version_str[9];

version = R_RSPI_GetVersion();

version_high = (version >> 16)&0xf;
version_low  =  version & 0xff;

sprintf(version_str, "RSPIv%1.1hu.%2.2hu", version_high, version_low);
```

4. Port Configuration

4.1 System Initialization

RSPI I/O signals may be optionally assigned to different I/O ports. The initialization of the Multi-function Pin Controller (MPC) and the PORT registers must be handled by the application, either through calls to the FIT MPC module, or directly. If the 3-wire interface mode is being used then a GPIO port must be configured to handle the Slave Select signal. GPIOs may be configured using the FIT GPIO module API, or through direct register settings.

The port initialization must be performed after the driver Open function is called. An example of direct initialization for the RSK-RX64M channel 0 is shown below.

```
/* Use BSP function to unlock the MPC registers. */
R_BSP_RegisterProtectDisable(BSP_REG_PROTECT_MPC);

/* SPI 4-wire mode. */
MPC.PC4PFS.BYTE = 0x0D; /* PC4 is SSLA0. */
PORTC.PMR.BIT.B4 = 1;      /* Set to peripheral mode. */

/* Make the RSPI peripheral signal assignment selections in the MPC registers. */
MPC.PC7PFS.BYTE = 0x0D; /* PC7 is MISOA */
MPC.PC6PFS.BYTE = 0x0D; /* PC6 is MOSIA */
MPC.PC5PFS.BYTE = 0x0D; /* PC5 is RSPCKA */

/* Set RSPI signal ports to peripheral mode. */
PORTC.PMR.BIT.B7 = 1;
PORTC.PMR.BIT.B6 = 1;
PORTC.PMR.BIT.B5 = 1;

/* Re-lock the MPC registers. */
R_BSP_RegisterProtectEnable(BSP_REG_PROTECT_MPC);
```

5. Driver Architecture

5.1 System Examples

The driver supports single-master/multi-slave mode operation, or slave-mode operation. Each RSPI channel controls one SPI bus. Multiple-master operation on the same bus is not supported in this driver. An example of a single master connected to multiple slaves on one SPI bus is shown.

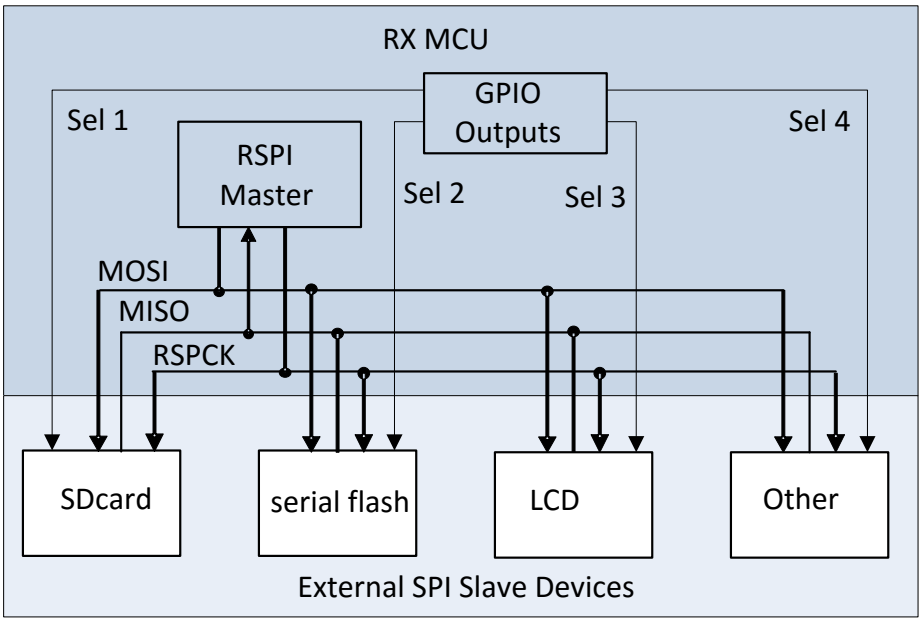

Figure 2. This example shows the use of GPIO ports to serve as the slave select signals (3-Wire mode).

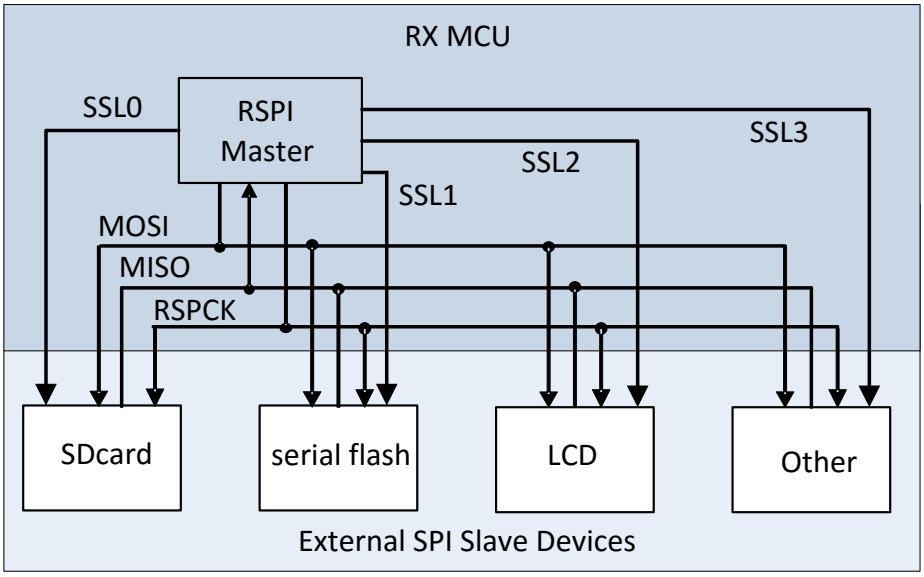

Figure 3. The built-in RSPI peripheral slave select hardware (SSL) may be used to generate the signals (SPI 4-Wire mode).

5.2 Multi-Channel RSPI Support

For supported RX family MCUs that have multiple channels of RSPI available, this driver will operate all available channels on an individually selectable basis with the same body of code. Each channel can be configured with its own setup independently of the other channels in use.

6. Data transfer operations

The RSPI driver provides three data transfer functions available for both master and slave mode operation; Write, Read, Write/Read (full duplex). All write and read operations are performed with the RSPI configured for full-duplex mode. All write and read operations are done in a non-blocking fashion where the call to the write or read function sets up the transfer then returns. The function returns as soon as the operation has been successfully initialized, or there is an error.

If locking has been enabled by the configuration option, the RSPI channel will be locked for the duration of the operation. After that, the remainder of the transfer operation is performed within RSPI interrupt handler routines.

An RSPI Transmit buffer empty interrupt (SPTI) ISR is implemented to perform the first transmit operation. When the first one or two SPTI interrupts occur (depending on Master-mode or Slave-mode respectively), the SPTI ISR disables further SPTI interrupts. After that the remaining transfer operation is processed through the receive buffer full interrupt (SPRI). This interrupt indicates that a complete frame of data has been received by RSPI, and that a complete frame has been clocked out.

SPTI and SPRI interrupts both call a common handler routine for all channels. The first time through, receive data can never be valid because nothing has been clocked out yet – it is just placing the first data into the transmit buffer. So, the receive data buffer is read to clear it and the data is discarded. Additionally, slave mode double buffers the slave transmit data so that back-to-back frames clocked by the master will not starve the slave's transmit shift register. In this case the SPTI interrupt will be entered twice before the SPRI interrupt takes over. The transfer process ends when the SPRI ISR is entered after the final data has been transferred.

When the specified number of frames has been transferred, the ISR terminates the operation, disables the RSPI channel and its interrupts, then calls the user-defined callback function to alert the application that it is done. If locking has been enabled then the channel is unlocked at this time.

6.1 Transmitting Data and Receiving Data

6.1.1 Transmitting data from RSPI

In Master mode, data is written by the RSPI Master on the MOSI (master out, slave in) line. In Slave mode, data is written by the RSPI Slave on the MISO (master in, slave out) line. Since all data transfer operations are internally performed in full duplex mode, when only transmitting the RSPI driver will read the receive data register to clear it, but will discard the result. Data to be transmitted is read from a buffer location pointed to by the user application, and it is copied to the RSPI transmit data register after being type-casted for the data type specified by the current operation.

6.1.2 RSPI Master receiving data from a SPI Slave

Data is received by the RSPI Master on the MISO (master in, slave out) line. With the RSPI peripheral configured as SPI bus Master, it is set for full-duplex operation in order to receive data from a slave device on the SPI bus. This requires the RSPI Master to output clocks to the Slave. Clocks are output only when the RSPI Master is sending data. Therefore, in order to read data from the SPI bus, the master must also simultaneously write data. This can either be actual data that needs to be transmitted (if the slave is capable of full-duplex communication), or it can be dummy data ignored by the slave. In this driver implementation the read data is clocked by dummy writes of a user definable data pattern.

6.1.3 RSPI Slave write to SPI Master

The Slave-mode write operation is nearly the same as the master-mode except that, after setting up for transmission, the slave waits for clocks from a master SPI device. Additionally, slave mode double buffers the slave transmit data so that back-to-back frames clocked by the master will not starve the slave's transmit shift register.

If not reading while writing, the read data register is cleared after every frame is transmitted. The transmit operation will terminate when the requested number of frames has been transmitted, or until aborted by user command.

6.1.4 RSPI Slave read from SPI Master

The Slave-mode read operation is exactly the same as the master-mode except that, after setting up for reception, the slave waits for clocks from a master SPI device. If not also transmitting valid data while receiving, the transmit data register is filled with a user definable dummy data pattern. The read operation will terminate when the requested number of frames has been received, or until aborted by user command.

6.2 Interrupts

6.2.1 Data transfer interrupts

The RSPI driver transmit and receive operations are performed in a non-blocking fashion; data transfer operations are carried out on an event driven basis with interrupt service routines. The RSPI Transmit Buffer Empty (SPTI), and Receive Buffer Full (SPRI) interrupts are used to call a common read/write function that performs a single frame receive and/or transmit procedure, depending on the state.

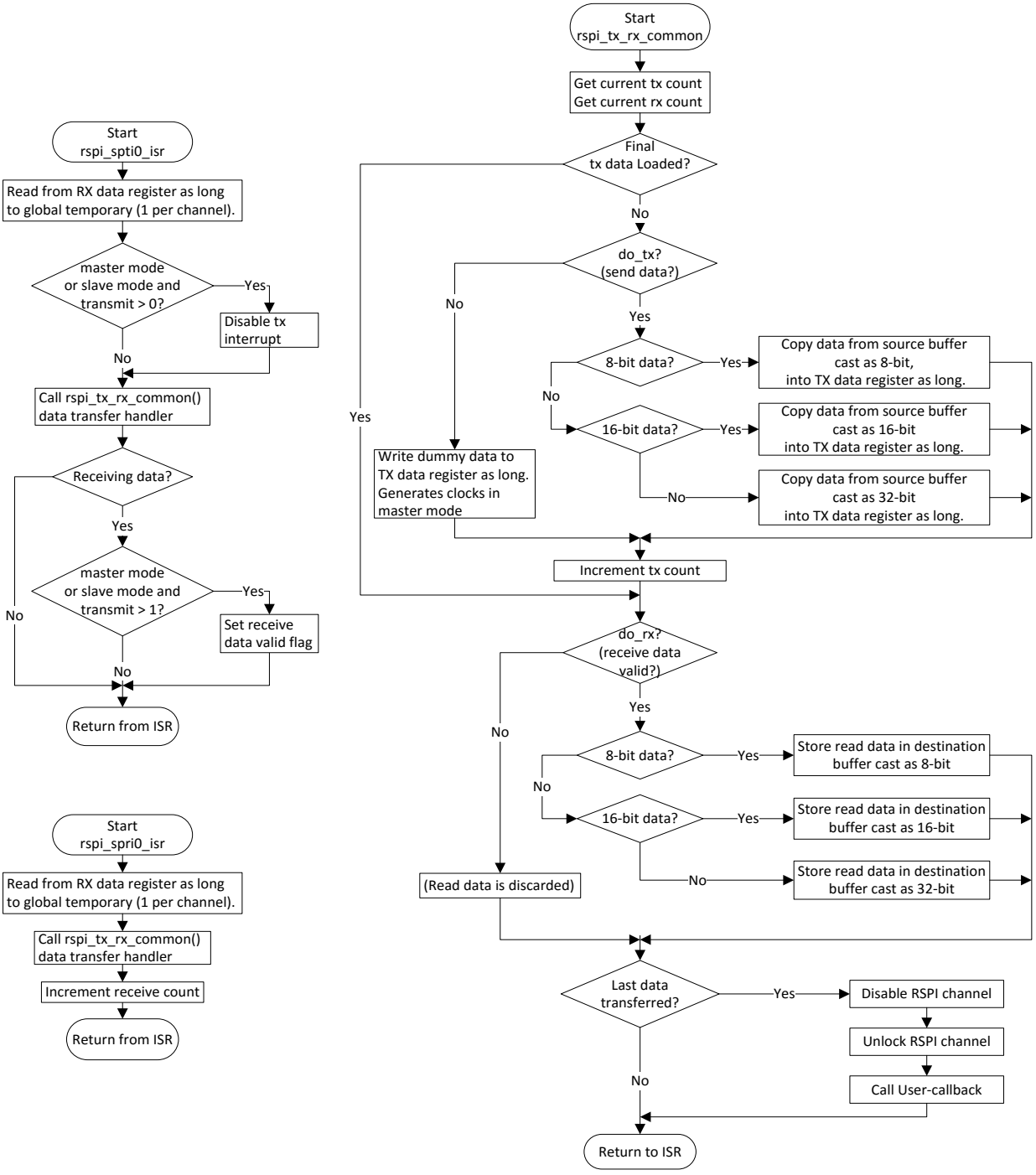

Figure 4. Common data transfer interrupt handler algorithm.

.

6.2.2 Error interrupts

The RSPI Error interrupts (SPEI) are used to call a common handler function that reads the status register to determine the interrupt cause. Further processing of the data transfer operation is halted and the callback function is called.
In the error interrupt handler processing, check each flag state for SPSR register in the order of OVRF→MODF→ UDRF→PERF.
Set the first-detected error flag state to argument event of callback function.

6.3 Callback Functions

The definition of callbacks follows the FIT 1.0 specification rules:

a. Callback functions take one argument. This argument is ‘void * pdata’.

b. Before calling a callback function the function pointer is checked to be valid. At a minimum the pointer is checked to be:

i. Non-null

ii. Not equal to FIT_NO_FUNC macro.

6.3.1 Example callback function prototype declaration.

```
void callback(void * pdata)
```

6.3.2 Invocation of Callback functions

At the conclusion of every transfer operation the user defined callback will be called. This will occur within the context of the interrupt handler that processed the transfer operation. Any error condition that generates an interrupt, most typically the receive-overrun error, will also call the callback. A pointer to a data structure containing the channel number and result code of the interrupt that calls the callback are be passed as the only argument. It is up to the user application to process the provided information appropriately. Since callbacks are being processed within the context of the interrupt, and interrupts are disabled at this time, it is strongly recommended that the user-defined callback function complete as quickly as possible to avoid missing further system interrupts.

The most typical use of the callback function is to inform the application that the data transfer has completed. This may be done by setting a “busy” flag just before starting the transfer, and then clearing the busy flag within the callback. When used in RTOS environments, then a semaphore or other flag or message service provided by the OS may be used within the callback.

Example transfer start:

```
/* Conditions: Channel currently open. */
g_transfer_complete = false;
rspi_result = R_RSPI_WriteRead(handle, my_command_word, source, dest, length);
if (RSPI_SUCCESS != rspi_result)
{
    return error;
}

while (!g_transfer_complete) // Poll for interrupt callback to set this.
{
    nop();// Do something useful while waiting for the transfer to complete.
}
```

Example callback function:

```
void my_callback(void * pdata)
{
    /* Examine the event to check for abnormal termination of transfer. */
    g_test_callback_event = (*(rspi_callback_data_t *)pdata).event_code;

    g_transfer_complete = true;
}
```

6.4 Relations of Data Output and RAM

Note that the data is not output in the order of the data stored in RAM when the data is 16-bit type or 32-bit type, and is Little endian. Perform byte swap processing as necessary. IP version RSPIc or later includes byte swap function.

6.4.1 Transmitting Data

(1) **16-bit type【Little endian】**

As shown in Figure 6-1, when 1 frame data is16-bit type, the data is inverted at the timing of writing RAM data to SPDR register. Therefore, the order of data output is as Byte1, Byte0, Byte3, Byte2…

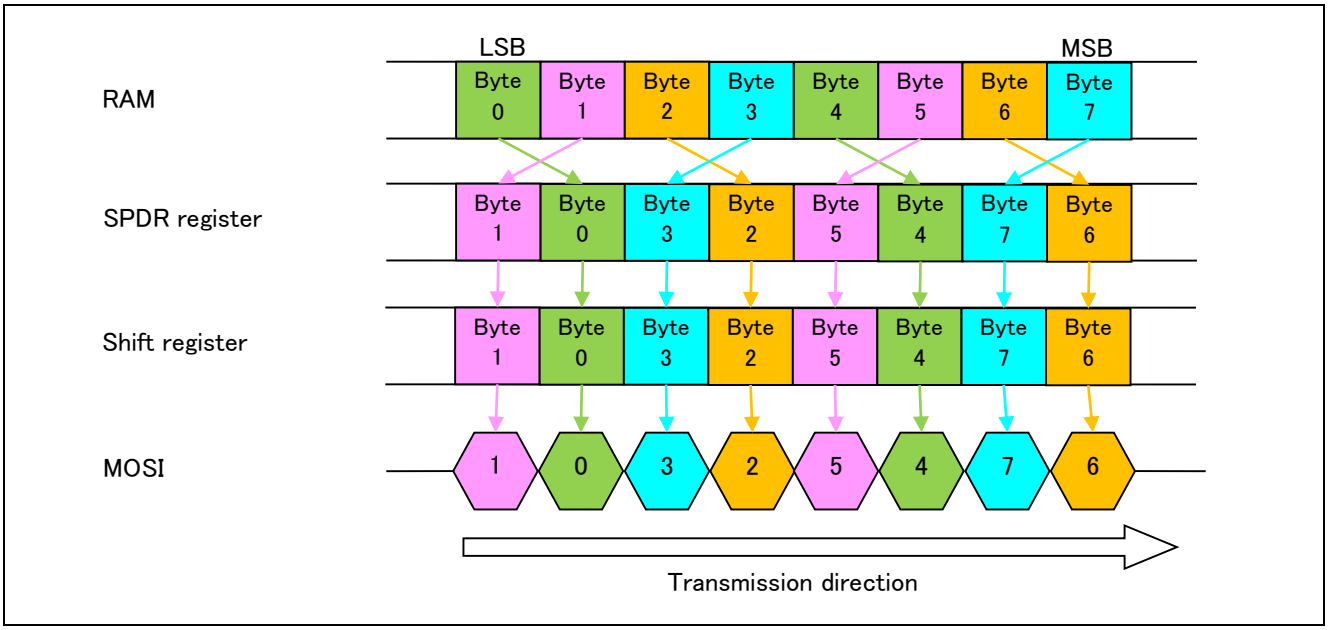

Figure 6-1 Transmitting Data 16-bit type【Little endian】 No byte swapping

IP version RSPIc or later includes byte swap function, and byte swap is available on the hardware. However, RSPI driver does not support 16-bit hardware byte swap.

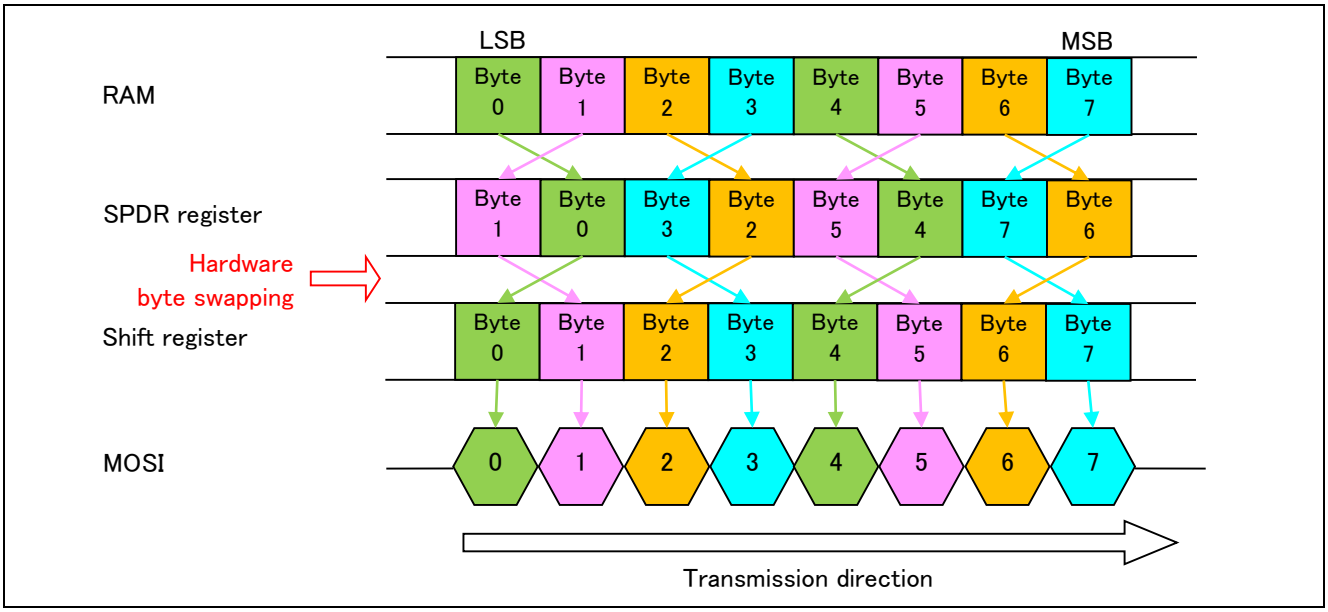

Figure 6-2 Transmitting Data 16-bit type【Little endian】 hardware byte swapping

(2) **32-bit type【Little endian】**

As shown in Figure 6-3, when 1 frame data is 32-bit type, the data is inverted at the timing of writing RAM data to SPDR register. Therefore, the order of data output is as Byte3, Byte2, Byte1, Byte0….

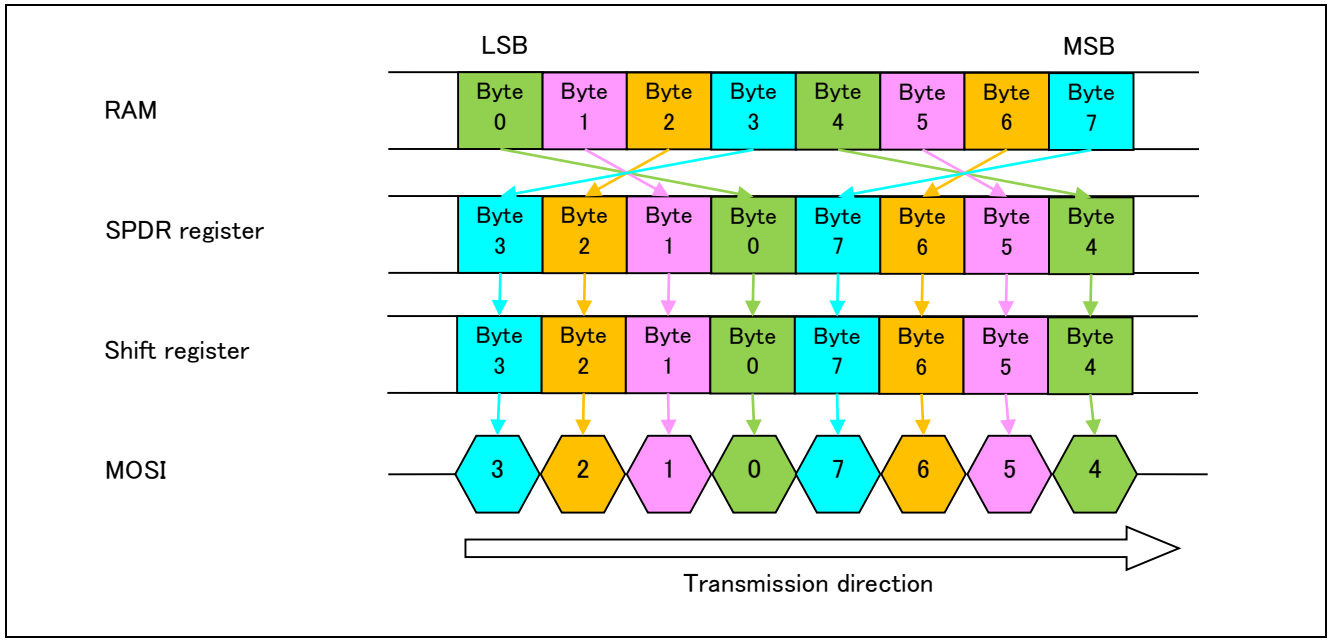

Figure 6-3 Transmitting Data 32-bit type【Little endian】No byte swapping

IP version RSPIc or later includes byte swap function, and byte swap is available on the hardware. RSPI driver supports 32-bit hardware byte swap.

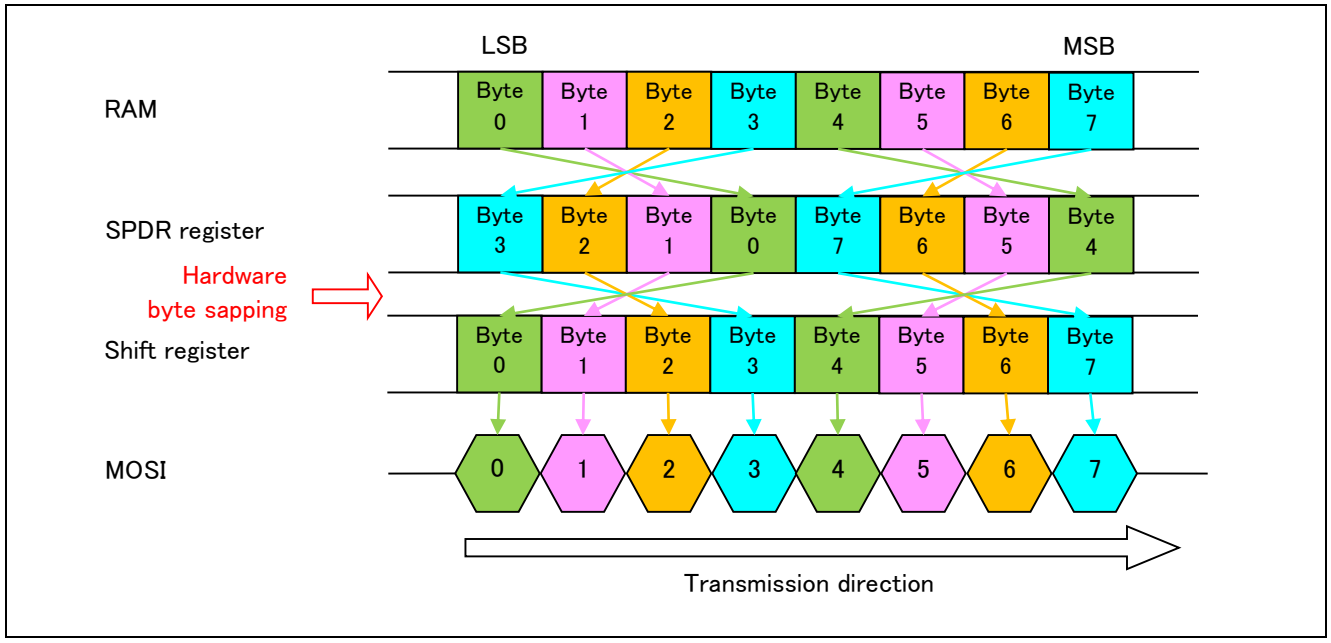

Figure 6-4 Transmitting Data 32-bit type【Little endian】hardware byte swapping

(3) **Other data type and Endian**

For the data type and endian shown below, the data is output in the order of the data stored in RAM.

- 8-bit type【Little endian/ Big endian】
- 16-bit type【Big endian】
- 32-bit type【Big endian】

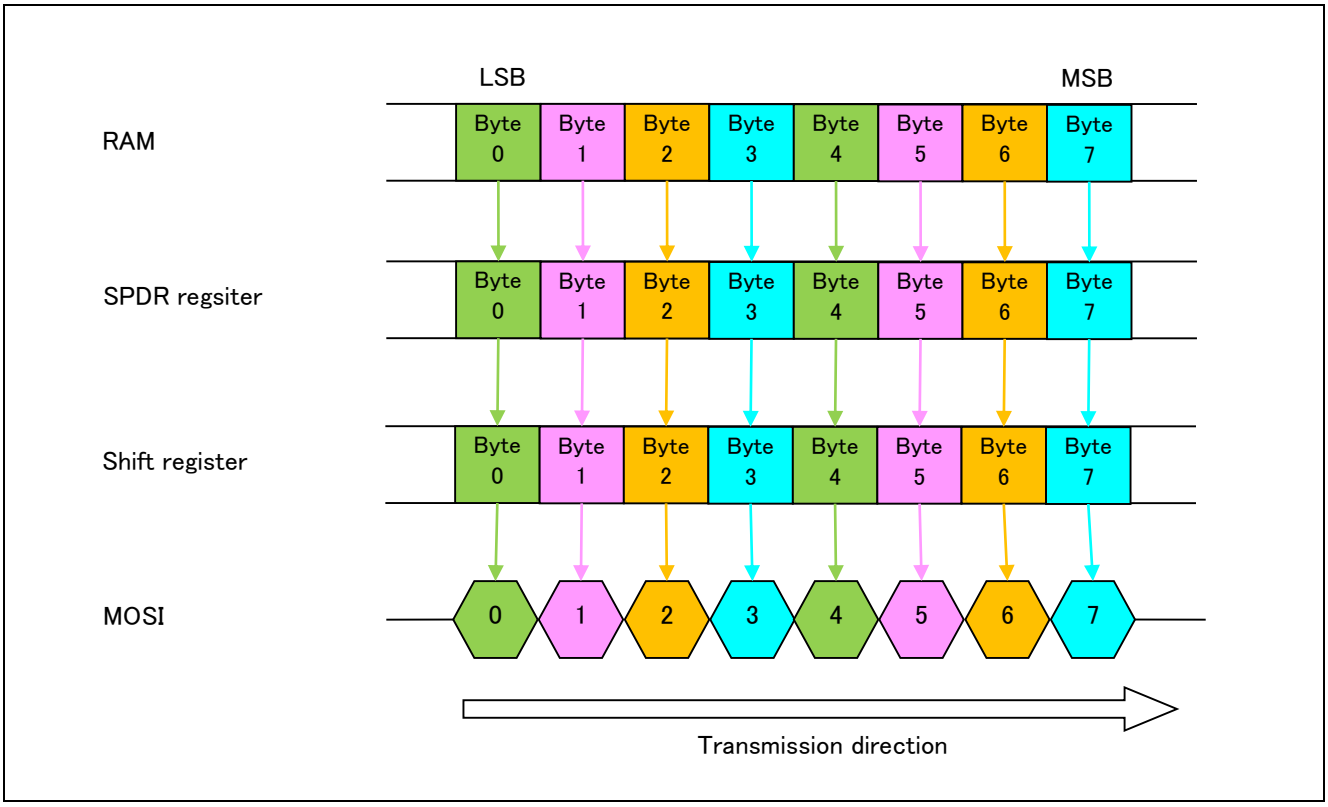

Figure 6-5 Transmitting Data Other data type and Endian

6.4.2 Receiving Data

(1) **16-bit type【Little Endian】**

As shown in Figure 6-6, when 1 frame data is 16-bit type, the data is inverted at the timing of reading the data from SPDR register to RAM. Therefore, the order of data stored in RAM is as Byte1, Byte0, Byte3, Byte2….

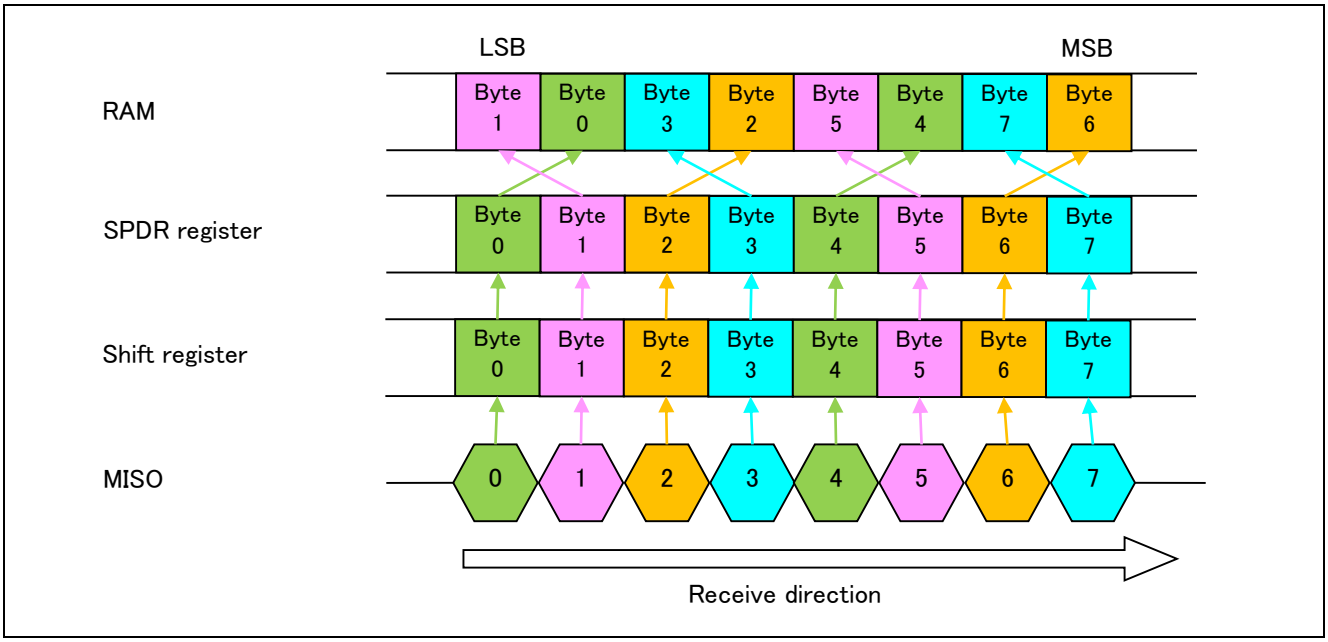

Figure 6-6 Receiving Data 16-bit type【Little endian】No byte swapping

IP version RSPIc or later includes byte swap function, and byte swap is available on the hardware.

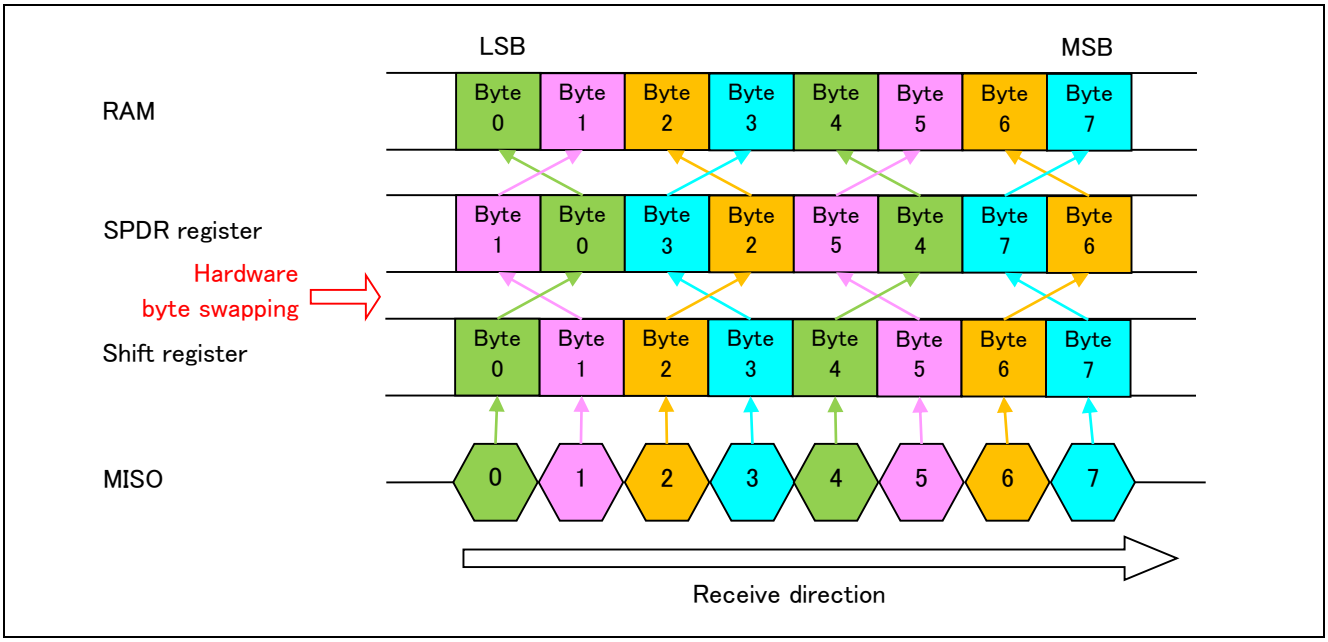

Figure 6-7 Receiving Data 16-bit type【Little endian】hardware byte swapping

(2) **32-bit type【Little endian】**

As shown in Figure 6-8, when 1 frame data is 32-bit type, the data is inverted at the timing of reading the data from SPDR register to RAM. Therefore, the order of the data stored in RAM is as Byte3, Byte2, Byte1, Byte0….

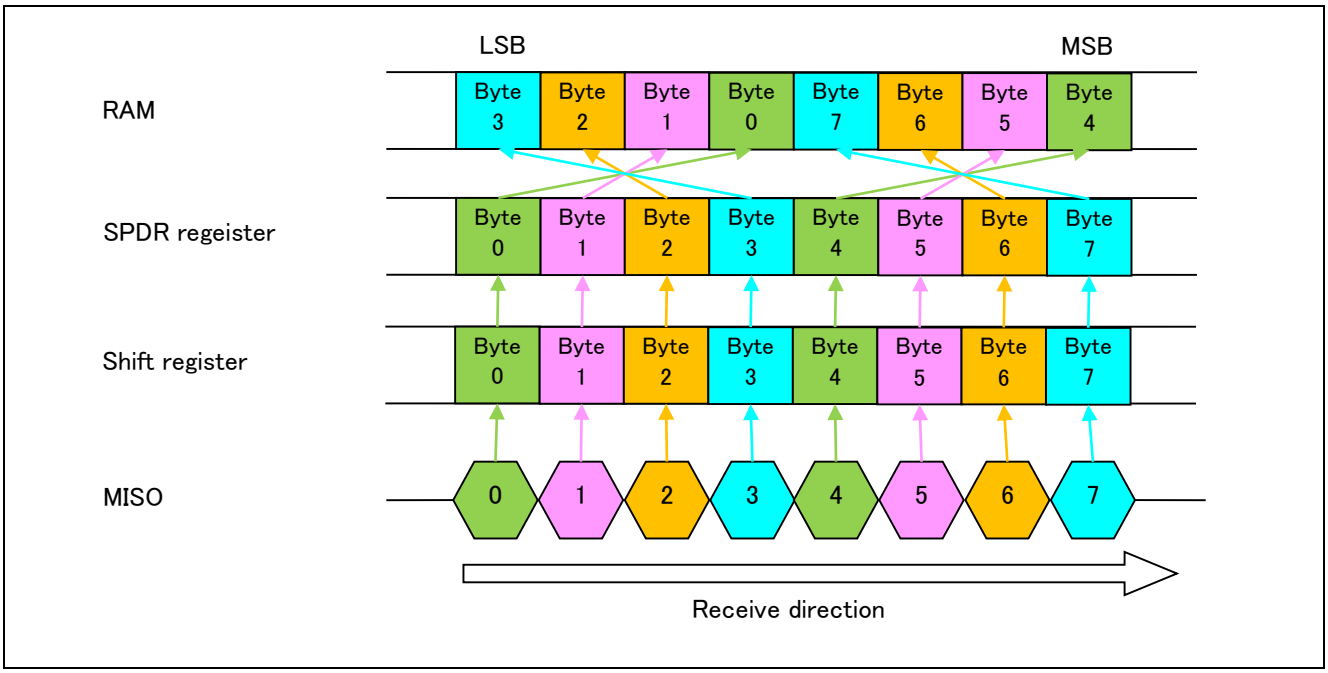

Figure 6-8 Receiving Data 32-bit type【Little endian】No byte swapping

IP version RSPIc or later includes byte swap function, and byte swap is available on the hardware.

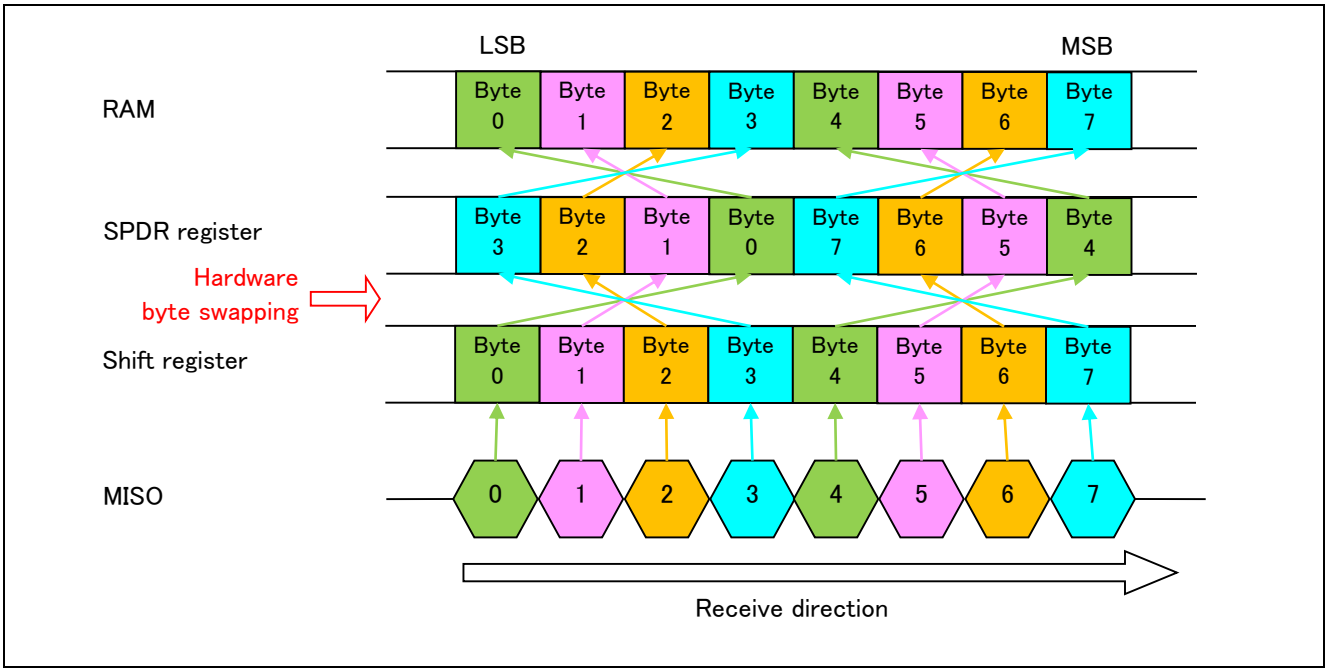

Figure 6-9 Receiving Data 32-bit type【Little endian】Hardware byte swapping

(3) **Other data type and Endian**

For the data type and endian shown below, the data is stored in the RAM in the order of data output.

- 8-bit type【Little endian/ Big endian】
- 16-bit type【Big endian】
- 32-bit type【Big endian】

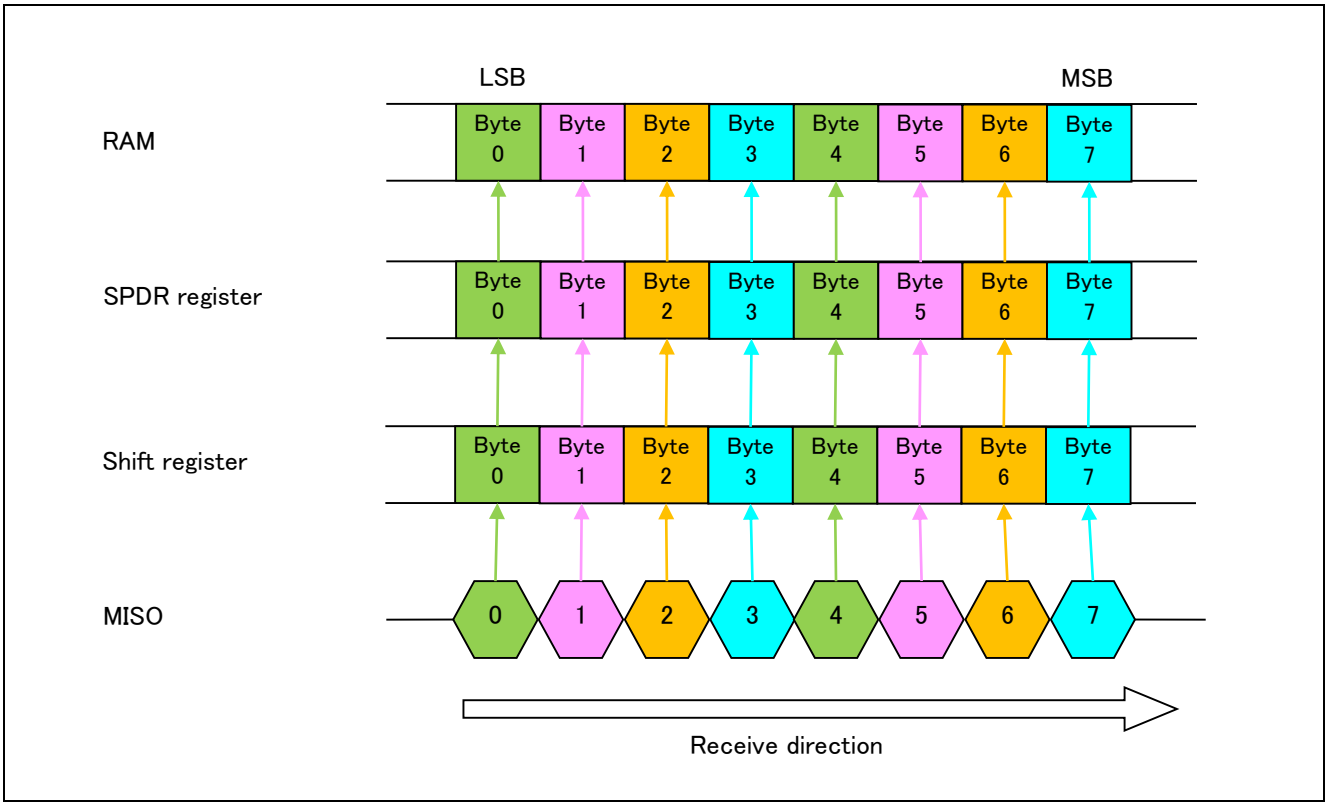

Figure 6-10 Receiving Data Other Data type and Endian

7. Demo Project

This application note includes one or more sample application projects to demonstrate basic usage of the FIT RSPI Module. The sample code is intended to provide a quick functional example of common API function calls in use.

The provided sample application simulates a full-duplex transfer (simultaneous transmit and receive) by routing the Master output data to the Master input data with a jumper wire. Data received is tested to confirm that it matches the data sent. The RSPI module version number is retrieved and can be displayed on the Renesas Virtual Debug Console window if desired.

7.1 Adding the Demo to a Workspace

Demo projects are found in the FITDemos subdirectory of the distribution file for this application note. Demo projects are MCU and board specific. Locate the project that matches the Renesas development board you will be using. To add a demo project to a workspace, select File>Import>General>Existing Projects into Workspace, then click “Next”. From the Import Projects dialog, choose the “Select archive file” radio button. “Browse” to the FITDemos subdirectory, select the desired demo zip file, then click “Finish”.

7.2 Running the demo

1. Prepare the board by jumpering MOSIA to MISOA depending on the target board:
   a) RSKRX113
      i) Connect expansion header J3 pin 24 to J3 pin 23.
   b) RSKRX64M and RSKRX71M
      i) Remove any jumper plugs from board jumpers J14 and J12.
      ii) Connect J14 pin 2 to J12 pin 2.
   c) RSKRX231
      i) Connect expansion header J3 pin 14 to J3 pin 13.
   d) RSKRX65N
      i) Connect expansion header J13 pin 2 to J11 pin 2.
2. Build and download the sample application to the RSK board using the e2studio debugger.
3. Select the Renesas Virtual Debug Console view in e2studio to view print information.
4. Run the application in the debugger.
5. Observe the version number print in the debug console window.
6. Observe the green LED illuminate to indicate successful transfer. If transfer fails the red LED will light.

Technical Update Information

The following technical update applies to this module.

- TN-RX*-A147A/E
  The Technical Update describes how to check all data transmission completion without using interrupt.
  The contents of the Technical Update does not apply to the RSPI driver as it uses interrupt on completion of transmission.

Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Nov 15, 2013	--	First edition issued
1.20	April 4, 2014	--	Updated list of supported/tested MCUs
1.30	Jan 20, 2015	--	Updated list of supported/tested MCUs
		8	Added Section 2.9 Code Size and RAM usage
		32	Added Demo Project section
			Corrected fonts
1.40	Jun 29, 2015	1,3,9,33	Updated to include support for RX231
1.50	Sep 30, 2016	1	Updated to include support for RX65N, RX130, RX230, RX23T and RX24T.
		16	Changed the chapter number of API Functions to 3 from 6.
		34	Added Section 6.4 Relations of Data Output and RAM

General Precautions in the Handling of MPU/MCU Products

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

1. Handling of Unused Pins

   Handle unused pins in accordance with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

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

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
3. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; and safety equipment etc.
   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (nuclear reactor control systems, military equipment etc.). You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application for which it is not intended. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technology may 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 should not use Renesas Electronics products or technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. When exporting the Renesas Electronics products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
11. This document may not be reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HALII Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 5.0