RX Family

RSCI Module Using Firmware Integration Technology

Introduction

This application note describes the enhanced serial communications interface (RSCI) module which uses Firmware Integration Technology (FIT). This module uses RSCI to provide Asynchronous, Synchronous, SPI (SSPI), and Manchester support for all channels of the RSCI peripheral. In this document, this module is referred to as the RSCI FIT module.

Target Devices

- RX26T Group (Products with 64 Kbytes of RAM)
- RX671 Group
- RX660 Group
- RX260, RX261 Group

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

Target Compilers

Renesas Electronics C/C++ Compiler Package for RX Family

GCC for Renesas RX

IAR C/C++ Compiler for Renesas RX

For details of the confirmed operation contents of each compiler, refer to “6.1 Confirmed Operation Environment".

Contents

1. Overview

1.1 RSCI FIT Module

The RSCI FIT module can be used by being implemented in a project as an API. See section 2.13, Adding the FIT Module to Your Project for details on methods to implement this FIT module into a project.

1.2 Overview of the RSCI FIT Module

RSCI can handle both asynchronous and clock synchronous serial communications. RSCI has FIFO buffer of 32 stages in transmission/reception blocks, and it can select the FIFO composition, and it can transmit/receive efficiently, and it can also communicate continuously.

Additionally, the driver supports the following features in Asynchronous mode:

- Noise cancellation
- Outputting baud clock on the SCK pin
- One-way flow control of either CTS or RTS

All basic UART, Master SPI, Master Synchronous, and Manchester mode functionality are supported by this driver.

Features not supported by this driver are:

- Extended
- Multiprocessor mode (all channels)
- Event linking
- DMAC/DTC data transfer
- RZI code

<u>Handling of Channels</u>

This is a multi-channel driver, and it supports all channels present on the peripheral. Specific channels can be excluded via compile-time defines to reduce driver RAM usage and code size if desired. These defines are specified in “r_rsci_rx_config.h”.

An individual channel is initialized in the application by calling R_RSCI_Open(). This function applies power to the peripheral and initializes settings particular to the specified mode. A handle is returned from this function to uniquely identify the channel. The handle references an internal driver structure that maintains pointers to the channel’s register set, buffers, and other critical information. It is also used as an argument for the other API functions.

<u>Interrupts, and Transmission and Reception</u>

Interrupts supported by this driver are TXI, TEI, RXI, and ERI. For Asynchronous mode, circular buffers are used to queue incoming as well as outgoing data. The size of these buffers can also be set on compilation.

The TXI and TEI interrupts are used in Asynchronous or Manchester mode. The TXI interrupt occurs when transmit data in the TDR register has been shifted into the TSR register. During this interrupt, the next byte in the transmit circular buffer is placed into the TDR register to be ready for transmit. If a callback function is provided in the R_RSCI_Open() call, it is called here with a TEI event passed to it. Support for TEI interrupts may be removed from the driver via a setting in “r_rsci_rx_config.h”.

The RXI interrupt occurs each time the RDAT field of the RDR register has shifted in receive data. In Asynchronous or Manchester mode, this byte is loaded into the receive circular buffer during the interrupt for access later via an R_RSCI_Receive() call at the application level. If a callback function is provided, it is called with a receive event. If the receive queue is full, it is called with a queue full event while the last received byte is not stored. In SSPI and Synchronous modes, the shifted-in byte is loaded directly into the receive buffer specified from the last R_RSCI_Receive() or R_RSCI_SendReceive() call. The data received before R_RSCI_Receive() or R_RSCI_SendReceive() call is ignored. With SSPI and Synchronous modes, data is transmitted and received in the RXI interrupt handler. The number of data remaining to be transferred or received can be checked with the value of the transmit counter (tx_cnt) and received counter (rx_cnt) in the handle set for the fourth parameter of the R_RSCI_Open function. Refer to 2.10, Parameters for details.

To use RSCI nested interrupts, enable macros RSCI_CFG_CHn_EN_TXI_NESTED_INT, RSCI_CFG_CHn_EN_RXI_NESTED_INT, RSCI_CFG_CHn_EN_TEI_NESTED_INT and RSCI_CFG_CHn_EN_ERI_NESTED_INT for each related channel.

Error Detection

The ERI interrupt occurs when a framing, overrun, or parity error is detected by the receive device. In Manchester mode, there are also Manchester code, preface, start bit, or receive Sync errors. If a callback function is provided, the interrupt determines which error occurred and notifies the application of the event. Refer to 2.12, Callback Function for details.

This FIT module clears the error flag in the ERI interrupt handler regardless of the callback function provided or not. If the FIFO function is enabled, the callback function is called before the error flag is cleared. So, the data where the error occurred can be determined by reading the RDR register for the number of data received. Refer to 2.12 Callback Function for details.

1.3 API Overview

Table 1.1 lists the API functions included in this module.

Table 1.1 API Functions

Function Name	Description
R_RSCI_Open()	Applies power to the RSCI channel, initializes the associated registers, enables interrupts, and provides the channel handle for use with other API functions. Specifies the callback function which is called when a receive error or other interrupt events occur.
R_RSCI_Close()	Removes power to the RSCI channel and disables the associated interrupts.
R_RSCI_Send()	Initiates transmit if transmitter is not in use.
R_RSCI_Receive()	For Asynchronous or Manchester mode, fetches data from a queue which is filled by RXI interrupts. For Synchronous and SSPI modes, initiates dummy data transmission and reception if transceiver is not in use.
R_RSCI_SendReceive()	For Synchronous and SSPI modes only. Transmits and receives data simultaneously if the transceiver is not in use.
R_RSCI_Control()	Handles special hardware or software operations for the RSCI channel.
R_RSCI_GetVersion()	Returns at runtime the driver version number.

1.4 Limitations

None.

1.5 Using the FIT RSCI module

1.5.1 Using FIT RSCI module in C++ project

For C++ project, add FIT RSCI module interface header file within extern “C”{}:

```
Extern “C”
{
  #include “r_smc_entry.h”
  #include “r_rsci_rx_if.h”
}
```

2. API Information

This FIT module has been confirmed to operate under the following conditions.

2.1 Hardware Requirements

The MCU used must support the following functions:

- RSCI
- GPIO

2.2 Software Requirements

This driver is dependent upon the following FIT module:

- Renesas Board Support Package (r_bsp) v6.10 or higher
- r_byteq (Asynchronous or Manchester mode)

2.3 Limitations

2.3.1 RAM Location Limitations

In FIT, if a value equivalent to NULL is set as the pointer argument of an API function, error might be returned due to parameter check. Therefore, do not pass a NULL equivalent value as pointer argument to an API function.

The NULL value is defined as 0 because of the library function specifications. Therefore, the above phenomenon would occur when the variable or function passed to the API function pointer argument is located at the start address of RAM (address 0x0). In this case, change the section settings or prepare a dummy variable at the top of the RAM so that the variable or function passed to the API function pointer argument is not located at address 0x0.

In the case of the CCRX project (e2 studio V21.7.0), the RAM start address is set as 0x4 to prevent the variable from being located at address 0x0. In the case of the GCC project (e2 studio V21.7.0) and IAR project (EWRX V4.20.1), the start address of RAM is 0x0, so the above measures are necessary.

The default settings of the section may be changed due to the IDE version upgrade. Please check the section settings when using the latest IDE.

2.4 Supported Toolchain

This driver has been confirmed to work with the toolchain listed in 6.1, Confirmed Operation Environment.

2.5 Interrupt Vector

The RXIn and ERIn interrupt is enabled by executing the R_RSCI_Open function.

For SSPI and synchronous modes, interrupts TXIn and TEIn are not used in these modes.

Table 2.1 lists the interrupt vector used in the RSCI FIT Module.

Table 2.1 Interrupt Vector Used in the RSCI FIT Module

Device	Interrupt Vector
RX671 RX660	RXI interrupt (vector no.: 32) TXI interrupt (vector no.: 33) RXI interrupt (vector no.: 42) TXI interrupt (vector no.: 43) GROUPAL0 interrupt (vector no.: 112) • TEI interrupt (group interrupt source no.: 24) • ERI interrupt (group interrupt source no.: 25) • TEI interrupt (group interrupt source no.: 27) • ERI interrupt (group interrupt source no.: 28)
RX26T	RXI interrupt (vector no.: 100) TXI interrupt (vector no.: 101) RXI interrupt (vector no.: 102) TXI interrupt (vector no.: 103) RXI interrupt (vector no.: 114) TXI interrupt (vector no.: 115) GROUPBL1 interrupt (vector no.: 111) • TEI interrupt (group interrupt source no.: 24) • ERI interrupt (group interrupt source no.: 25) • TEI interrupt (group interrupt source no.: 26) • ERI interrupt (group interrupt source no.: 27) GROUPAL0 interrupt (vector no.: 112) • TEI interrupt (group interrupt source no.: 12) • ERI interrupt (group interrupt source no.: 13)
RX260 RX261	ERI interrupt (vector no.: 214) RXI interrupt (vector no.: 215) TXI interrupt (vector no.: 216) TEI interrupt (vector no.: 217) ERI interrupt (vector no.: 230) RXI interrupt (vector no.: 231) TXI interrupt (vector no.: 232) TEI interrupt (vector no.: 233) ERI interrupt (vector no.: 234) RXI interrupt (vector no.: 235) TXI interrupt (vector no.: 236) TEI interrupt (vector no.: 237)

2.6 Header Files

All API calls and their supporting interface definitions are located in r_rsci_rx_if.h.

2.7 Integer Types

This project uses ANSI C99. These types are defined in stdint.h.

2.8 Configuration Overview

The configuration option settings of this module are located in r_rsci_rx_config.h. The option names and setting values are listed in the table below:

Configuration options in r_rsci_rx_config.h	
RSCI_CFG_PARAM_CHECKING_ENABLE 1	1: Parameter checking is included in the build. 0: Parameter checking is omitted from the build. Setting this #define to BSP_CFG_PARAM_CHECKING_ENABLE utilizes the system default setting.
RSCI_CFG_ASYNC_INCLUDED 1 RSCI_CFG_SYNC_INCLUDED 0 RSCI_CFG_SSPI_INCLUDED 0 RSCI_CFG_MANC_INCLUDED 0	These #defines are used to include code specific to their mode of operation. A value of 1 means that the supporting code will be included. Use a value of 0 for unused modes to reduce overall code size.
RSCI_CFG_DUMMY_TX_BYTE 0xFF	This #define is used only with SSPI and Synchronous mode. It is the value of dummy data which is clocked out for each byte clocked in during the R_RSCI_Receive() function call.
RSCI_CFG_CH0_INCLUDED 0 RSCI_CFG_CH8_INCLUDED 0 RSCI_CFG_CH9_INCLUDED 0 RSCI_CFG_CH10_INCLUDED 0 RSCI_CFG_CH11_INCLUDED 0	Each channel has associated with it transmit and receive buffers, counters, interrupts, and other program and RAM resources. Setting a #define to 1 allocates resources for that channel. Be sure to enable the channels you will be using in the config file.
RSCI_CFG_CH0_TX_BUFSIZ 80 RSCI_CFG_CH8_TX_BUFSIZ 80 RSCI_CFG_CH9_TX_BUFSIZ 80 RSCI_CFG_CH10_TX_BUFSIZ 80 RSCI_CFG_CH11_TX_BUFSIZ 80	These #defines specify the size of the buffer to be used in Asynchronous or Manchester mode for the transmit queue on each channel. If the corresponding RSCI_CFG_CHn_INCLUDED is set to 0, RSCI_CFG_ASYNC_INCLUDED is set to 0, or RSCI_CFG_MANC_INCLUDED is set to 0, the buffer is not allocated.
RSCI_CFG_CH0_RX_BUFSIZ 80 RSCI_CFG_CH8_RX_BUFSIZ 80 RSCI_CFG_CH9_RX_BUFSIZ 80 RSCI_CFG_CH10_RX_BUFSIZ 80 RSCI_CFG_CH11_RX_BUFSIZ 80	These #defines specify the size of the buffer to be used in Asynchronous or Manchester mode for the receive queue on each channel. If the corresponding RSCI_CFG_CHn_INCLUDED is set to 0, RSCI_CFG_ASYNC_INCLUDED is set to 0, or RSCI_CFG_MANC_INCLUDED is set to 0, the buffer is not allocated.
RSCI_CFG_TEI_INCLUDED 0	Setting this #define to 1 causes the Transmit Buffer Empty interrupt code to be included. This interrupt occurs when the last bit of the last byte of data has been sent. The interrupt calls the user's callback function (specified in R_RSCI_Open()) and passes it an RSCI_EVT_TEI event.
RSCI_CFG_ERI_TEI_PRIORITY 3	This sets the receiver error interrupt (ERI) and transmit end interrupt (TEI) priority level. 1 is the lowest priority and 15 is the highest. The ERI interrupt handles overrun, framing, and parity errors for all channels. In Manchester mode, there are also Manchester code, preface, start bit, and receive Sync errors. The TEI interrupt indicates when the last bit has been transmitted and the transmitter is idle (Asynchronous/Manchester mode).

Configuration options in r_rsci_rx_config.h	
RSCI_CFG_CH0_EN_TXI_NESTED_INT 0 RSCI_CFG_CH8_EN_TXI_NESTED_INT 0 RSCI_CFG_CH9_EN_TXI_NESTED_INT 0 RSCI_CFG_CH10_EN_TXI_NESTED_INT 0 RSCI_CFG_CH11_EN_TXI_NESTED_INT 0	Specifies whether to include code for nested interrupt TXI Enable =1, Disable =0.
RSCI_CFG_CH0_EN_RXI_NESTED_INT 0 RSCI_CFG_CH8_EN_RXI_NESTED_INT 0 RSCI_CFG_CH9_EN_RXI_NESTED_INT 0 RSCI_CFG_CH10_EN_RXI_NESTED_INT 0 RSCI_CFG_CH11_EN_RXI_NESTED_INT 0	Specifies whether to include code for nested interrupt RXI Enable =1, Disable =0.
RSCI_CFG_CH0_EN_TEI_NESTED_INT 0 RSCI_CFG_CH8_EN_TEI_NESTED_INT 0 RSCI_CFG_CH9_EN_TEI_NESTED_INT 0 RSCI_CFG_CH10_EN_TEI_NESTED_INT 0 RSCI_CFG_CH11_EN_TEI_NESTED_INT 0	Specifies whether to include code for nested interrupt TEI Enable =1, Disable =0.
RSCI_CFG_CH0_EN_ERI_NESTED_INT 0 RSCI_CFG_CH8_EN_ERI_NESTED_INT 0 RSCI_CFG_CH9_EN_ERI_NESTED_INT 0 RSCI_CFG_CH10_EN_ERI_NESTED_INT 0 RSCI_CFG_CH11_EN_ERI_NESTED_INT 0	Specifies whether to include code for nested interrupt ERI Enable =1, Disable =0.
RSCI_CFG_CH10_FIFO_INCLUDED 0 RSCI_CFG_CH11_FIFO_INCLUDED 0	1: Processing regarding the FIFO function is included in the build 0: processing regarding the FIFO function is omitted from the build
RSCI_CFG_CH10_TX_FIFO_THRESH 8 RSCI_CFG_CH11_TX_FIFO_THRESH 8	When the RSCI operating mode is clock synchronous mode or simple SPI mode, set the values same as the receive FIFO threshold value. 0 to 31: Specifies the threshold value of the transmit FIFO.
RSCI_CFG_CH10_RX_FIFO_THRESH 8 RSCI_CFG_CH11_RX_FIFO_THRESH 8	1 to 31: Specifies the threshold value of the receive FIFO.
RSCI_CFG_CH0_DATA_MATCH_INCLUDED 0 RSCI_CFG_CH8_DATA_MATCH_INCLUDED 0 RSCI_CFG_CH9_DATA_MATCH_INCLUDED 0 RSCI_CFG_CH10_DATA_MATCH_INCLUDED 0 RSCI_CFG_CH11_DATA_MATCH_INCLUDED 0	1: Processing regarding the data match function is included in the build 0: processing regarding the data match function is omitted from the build
RSCI_CFG_CH0_TX_SIGNAL_TRANSITION_TIMING_INCLUDED 0 RSCI_CFG_CH8_TX_SIGNAL_TRANSITION_TIMING_INCLUDED 0 RSCI_CFG_CH9_TX_SIGNAL_TRANSITION_TIMING_INCLUDED 0 RSCI_CFG_CH10_TX_SIGNAL_TRANSITION_TIMING_INCLUDED 0 RSCI_CFG_CH11_TX_SIGNAL_TRANSITION_TIMING_INCLUDED 0	Disable or enable Transmit signal transition timing adjustment feature Enable =1, Disable =0.
RSCI_CFG_CH0_RX_SIGNAL_TRANSITION_TIMING_INCLUDED 0 RSCI_CFG_CH8_RX_SIGNAL_TRANSITION_TIMING_INCLUDED 0 RSCI_CFG_CH9_RX_SIGNAL_TRANSITION_TIMING_INCLUDED 0 RSCI_CFG_CH10_RX_DATA_SAMPLING_TIMING_INCLUDED 0 RSCI_CFG_CH11_RX_DATA_SAMPLING_TIMING_INCLUDED 0	Disable or enable Receive data sampling timing adjust feature Enable =1, Disable =0.

Configuration options in r_rsci_rx_config.h	
RSCI_CFG_CH9_TX_SEL_ENCODING_POL 0 RSCI_CFG_CH10_TX_SEL_ENCODING_POL 0 RSCI_CFG_CH11_TX_SEL_ENCODING_POL 0	0: Logic 0 is encoded as a low to high transition and logic 1 is encoded as a high to low transition. 1: Logic 0 is encoded as a high to low transition and logic 1 is encoded as a low to high transition.
RSCI_CFG_CH9_RX_SEL_DECODING_POL 0 RSCI_CFG_CH10_RX_SEL_DECODING_POL 0 RSCI_CFG_CH11_RX_SEL_DECODING_POL 0	0: Low to high transition is decoded to logic 0 and high to low transition is decoded to logic 1. 1: high to low transition is decoded to logic 0 and low to high transition is decoded to logic 1.
RSCI_CFG_CH9_TX_PREFACE_LENGTH 8 RSCI_CFG_CH10_TX_PREFACE_LENGTH 8 RSCI_CFG_CH11_TX_PREFACE_LENGTH 8	0 to 15: Specifies the preface length value of the transmit.
RSCI_CFG_CH9_RX_PREFACE_LENGTH 8 RSCI_CFG_CH10_RX_PREFACE_LENGTH 8 RSCI_CFG_CH11_RX_PREFACE_LENGTH 8	0 to 15: Specifies the preface length value of the receive.
RSCI_CFG_CH9_TX_PREFACE_PATTERN 0 RSCI_CFG_CH10_TX_PREFACE_PATTERN 0 RSCI_CFG_CH11_TX_PREFACE_PATTERN 0	0 to 3: Specifies the preface pattern value of the transmit.
RSCI_CFG_CH9_RX_PREFACE_PATTERN 0 RSCI_CFG_CH10_RX_PREFACE_PATTERN 0 RSCI_CFG_CH11_RX_PREFACE_PATTERN 0	0 to 3: Specifies the preface pattern value of the receive.

2.9 Code Size

Typical code sizes associated with this module are listed below.

The ROM (code and constants) and RAM (global data) sizes are determined by the build-time configuration options described in 2.8, Configuration Overview. The table lists reference values when the C compiler's compile options are set to their default values, as described in 2.4, Supported Toolchain. The compile option default values are optimization level: 2, optimization type: for size, and data endianness: little-endian. The code size varies depending on the C compiler version and compile options.

ROM and RAM minimum sizes (bytes)					
Device	Category		Memory usage		Remarks
			Renesas Compiler		
			With Parameter Checking	Without Parameter Checking	
RX671	Asynchronous mode	ROM	3472 bytes	3122 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	2990 bytes	2596 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	3744 bytes	3368 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	4550 bytes	4070 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		68 bytes		
	FIFO mode + Asynchronous mode	ROM	4372 bytes	3917 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	4024 bytes	3571 bytes	1 channel used
		RAM	44 bytes	44 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	5902 bytes	5362 bytes	Total 2 channels used
		RAM	408 bytes	408 bytes	Total 2 channels used
	Maximum stack usage		68 bytes		
RX660	Asynchronous mode	ROM	3497 bytes	3173 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	3019 bytes	2646 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	3610 bytes	3242 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	4458 bytes	4002 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		72 bytes		
	FIFO mode + Asynchronous mode	ROM	4454 bytes	4080 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	4100 bytes	3667 bytes	1 channel used
		RAM	44 bytes	44 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	5868 bytes	5352 bytes	Total 2 channels used
		RAM	408 bytes	408 bytes	Total 2 channels used

ROM and RAM minimum sizes (bytes)					
Device	Category		Memory usage		Remarks
			Renesas Compiler		
			With Parameter Checking	Without Parameter Checking	
	Maximum stack usage		72 bytes		
RX26T	Asynchronous mode	ROM	3628 bytes	3303 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	3128 bytes	2762 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	3740 bytes	3372 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	4621 bytes	4161 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		72 bytes		
	FIFO mode + Asynchronous mode	ROM	4574 bytes	4199 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	4208 bytes	3771 bytes	1 channel used
		RAM	44 bytes	44 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	6020 bytes	5500 bytes	Total 2 channels used
		RAM	408 bytes	408 bytes	Total 2 channels used
	Maximum stack usage		72 bytes		
RX260	Asynchronous mode	ROM	3382 bytes	3057 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	2912 bytes	2544 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	3498 bytes	3129 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	4331 bytes	3879 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		72 bytes		
RX261	Asynchronous mode	ROM	3382 bytes	3057 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	2912 bytes	2544 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	3498 bytes	3129 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	4331 bytes	3879 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		72 bytes		72 bytes

ROM and RAM minimum sizes (bytes)					
Device	Category		Memory usage		Remarks
			GCC Compiler		
			With Parameter Checking	Without Parameter Checking	
RX671	Asynchronous mode	ROM	6704 bytes	6016 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	5604 bytes	4883 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	7008 bytes	6264 bytes	1 channel used
		RAM	256 bytes	256 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	8892 bytes	7916 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		-		
	FIFO mode + Asynchronous mode	ROM	8408 bytes	7624 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	7636 bytes	6756 bytes	1 channel used
		RAM	44 bytes	44 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	11516 bytes	10420 bytes	Total 2 channels used
		RAM	408 bytes	408 bytes	Total 2 channels used
	Maximum stack usage		-		
RX660	Asynchronous mode	ROM	6604 bytes	5940 bytes	1 channel used
		RAM	256 bytes	256 bytes	1 channel used
	Clock synchronous mode	ROM	5760 bytes	5024 bytes	1 channel used
		RAM	0 bytes	0 bytes	1 channel used
	Manchester mode	ROM	6816 bytes	6072 bytes	1 channel used
		RAM	256 bytes	256 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	8496 bytes	7544 bytes	Total 2 channels used
		RAM	384 bytes	384 bytes	Total 2 channels used
	Maximum stack usage		-		
	FIFO mode + Asynchronous mode	ROM	8380 bytes	7604 bytes	1 channel used
		RAM	256 bytes	256 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	7856 bytes	6968 bytes	1 channel used
		RAM	128 bytes	128 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	11184 bytes	10104 bytes	Total 2 channels used
		RAM	384 bytes	384 bytes	Total 2 channels used
	Maximum stack usage		-		

ROM and RAM minimum sizes (bytes)					
Device	Category		Memory usage		Remarks
			GCC Compiler		
			With Parameter Checking	Without Parameter Checking	
RX26T	Asynchronous mode	ROM	4336 bytes	3864 bytes	1 channel used
		RAM	256 bytes	256 bytes	1 channel used
	Clock synchronous mode	ROM	3676 bytes	3140 bytes	1 channel used
		RAM	128 bytes	128 bytes	1 channel used
	Manchester mode	ROM	4552 bytes	3992 bytes	1 channel used
		RAM	256 bytes	256 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	5476 bytes	4788 bytes	Total 2 channels used
		RAM	384 bytes	384 bytes	Total 2 channels used
	Maximum stack usage		-		
	FIFO mode + Asynchronous mode	ROM	5560 bytes	4968 bytes	1 channel used
		RAM	256 bytes	256 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	5036 bytes	4372 bytes	1 channel used
		RAM	128 bytes	128 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	7228 bytes	6412 bytes	Total 2 channels used
		RAM	512 bytes	512 bytes	Total 2 channels used
	Maximum stack usage		-		
RX260	Asynchronous mode	ROM	4100 bytes	3644 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	3476 bytes	2948 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	4332 bytes	3780 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	5236 bytes	4540 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		-		
RX261	Asynchronous mode	ROM	4100 bytes	3636 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	3476 bytes	2940 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	4332 bytes	3772 bytes	1 channel used
		RAM	200 bytes	200 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	5236 bytes	4532 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		-		-

ROM and RAM minimum sizes (bytes)					
Device	Category		Memory usage		Remarks
			IAR Compiler		
			With Parameter Checking	Without Parameter Checking	
RX671	Asynchronous mode	ROM	5494 bytes	4874 bytes	1 channel used
		RAM	581 bytes	581 bytes	1 channel used
	Clock synchronous mode	ROM	4404 bytes	3793 bytes	1 channel used
		RAM	40 bytes	40 bytes	1 channel used
	Manchester mode	ROM	5805 bytes	5049 bytes	1 channel used
		RAM	589 bytes	589 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	7010 bytes	6154 bytes	Total 2 channels used
		RAM	781 bytes	781 bytes	Total 2 channels used
	Maximum stack usage		152 bytes		
	FIFO mode + Asynchronous mode	ROM	6751 bytes	6034 bytes	1 channel used
		RAM	589 bytes	589 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	5905 bytes	5173 bytes	1 channel used
		RAM	48 bytes	48 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	8897 bytes	7924 bytes	Total 2 channels used
		RAM	797 bytes	797 bytes	Total 2 channels used
	Maximum stack usage		224 bytes		
RX660	Asynchronous mode	ROM	5398 bytes	4807 bytes	1 channel used
		RAM	577 bytes	577 bytes	1 channel used
	Clock synchronous mode	ROM	4403 bytes	3871 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	5676 bytes	4957 bytes	1 channel used
		RAM	585 bytes	585 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	6690 bytes	5879 bytes	Total 2 channels used
		RAM	777 bytes	777 bytes	Total 2 channels used
	Maximum stack usage		152 bytes		
	FIFO mode + Asynchronous mode	ROM	6711 bytes	6018 bytes	1 channel used
		RAM	585 bytes	585 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	5936 bytes	5302 bytes	1 channel used
		RAM	44 bytes	44 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	8632 bytes	7715 bytes	Total 2 channels used
		RAM	793 bytes	793 bytes	Total 2 channels used
	Maximum stack usage		228 bytes		

ROM and RAM minimum sizes (bytes)					
Device	Category		Memory usage		Remarks
			IAR Compiler		
			With Parameter Checking	Without Parameter Checking	
RX26T	Asynchronous mode	ROM	5356 bytes	4760 bytes	1 channel used
		RAM	577 bytes	577 bytes	1 channel used
	Clock synchronous mode	ROM	4453 bytes	3853 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	5579 bytes	4914 bytes	1 channel used
		RAM	585 bytes	585 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	6811 bytes	5991 bytes	Total 2 channels used
		RAM	777 bytes	777 bytes	Total 2 channels used
	Maximum stack usage		152 bytes		
	FIFO mode + Asynchronous mode	ROM	6793 bytes	6104 bytes	1 channel used
		RAM	585 bytes	585 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	6093 bytes	5386 bytes	1 channel used
		RAM	44 bytes	44 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode	ROM	8755 bytes	7852 bytes	Total 2 channels used
		RAM	793 bytes	793 bytes	Total 2 channels used
	Maximum stack usage		228 bytes		
RX260	Asynchronous mode	ROM	5338 bytes	4723 bytes	1 channel used
		RAM	576 bytes	576 bytes	1 channel used
	Clock synchronous mode	ROM	4408 bytes	3808 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	5582 bytes	4913 bytes	1 channel used
		RAM	584 bytes	584 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	6434 bytes	5596 bytes	Total 2 channels used
		RAM	580 bytes	580 bytes	Total 2 channels used
	Maximum stack usage		148 bytes		
RX261	Asynchronous mode	ROM	5220 bytes	4605 bytes	1 channel used
		RAM	576 bytes	576 bytes	1 channel used
	Clock synchronous mode	ROM	4294 bytes	3694 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Manchester mode	ROM	5466 bytes	4797 bytes	1 channel used
		RAM	584 bytes	584 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	6316 bytes	5478 bytes	Total 2 channels used
		RAM	580 bytes	580 bytes	Total 2 channels used
	Maximum stack usage		148 bytes		

RAM requirements vary based on the number of channels configured. Each channel has associated data structures in RAM. In addition, for Asynchronous or Manchester mode, each Async or Manc channel will have a Transmit queue and a Receive queue. The buffers for these queues each have a minimum size of 2 bytes, or a total of 4 bytes per channel. Since the queue buffer sizes are user configurable, the RAM requirement will be increased or decreased directly by the amount allocated for buffers.
The formula for calculating Async or Manc mode RAM requirements is:

Number of channels used (1 to 2) × (Data structure per channel (32 bytes)
+ Transmit queue buffer size (size specified by RSCI_CFG_CHn_TX_BUFSIZ)
+ Receive queue buffer size (size specified by RSCI_CFG_CHn_RX_BUFSIZ))

* For FIFO mode, the data structure per channel is 36 bytes.

The Sync and SPI mode RAM requirements are number of channels × data structure per channel (fixed at 36 bytes, for FIFO mode, fixed at 40 bytes).

The ROM requirements vary based on the number of channels configured for use. The exact amount varies depending on the combination of channels selected and the effects of compiler code optimization.

2.10 Parameters

This section describes the parameter structure used by the API functions in this module. The structure is located in r_rsci_rx_if.h as are the prototype declarations of API functions.

Structure for Managing Channels

This structure is to store management information required to control RSCI channels. The contents of the structure vary depending on settings of the configuration option and the device used. Though the user does not need to care for the contents of the structure, if clock synchronous mode/SSPI mode is used, the number of data to be processed can be checked with tx_cnt or rx_cnt.

The following shows an example of the structure for RX671:

```
typedef struct st_rsci_ch_ctrl      // Channel management structure
{
rsci_ch_rom_t const *rom;     // Start address of the RSCI register for the
channel
rsci_mode_t mode;       // RSCI operating mode currently set for the channel
uint32_t baud_rate;     // Baud rate currently set for the channel
void (*callback)(void *p_args);     // Address of the callback function
union
{
#if (RSCI_CFG_ASYNC_INCLUDED || RSCI_CFG_MANC_INCLUDED)
byteq_hdl_t que;        // Transmit byte queue (asynchronous/manchester mode)
#endif
uint8_t *buf;     // Start address of the transmit buffer
//(clock synchronous/SSPI mode)
} u_tx_data;
union
{
#if (RSCI_CFG_ASYNC_INCLUDED || RSCI_CFG_MANC_INCLUDED)
byteq_hdl_t que;        // Receive byte queue (asynchronous/manchester mode)
#endif
uint8_t *buf;           // Start address of the receive buffer
                        //(synchronous/SSPI mode)
} u_rx_data;
bool tx_idle;           // Transmission idle state (idle state/transmitting)
#if (RSCI_CFG_SSPI_INCLUDED || RSCI_CFG_SYNC_INCLUDED)
bool save_rx_data;            // Receive data storage (enable/disable)
uint16_t tx_cnt;        // Transmit counter
uint16_t rx_cnt;        // Receive counter
bool tx_dummy;          // Transmit dummy data (enable/disable)
#endif
uint32_t pclk_speed;    // Operating frequency of the peripheral module clock
#if RSCI_CFG_FIFO_INCLUDED
uint8_t fifo_ctrl;            // FIFO function (enable/disable)
uint8_t rx_dflt_thresh;       // Recive FIFO threshold value (default)
uint8_t rx_curr_thresh;       // Recive FIFO threshold value (current)
uint8_t tx_dflt_thresh;       // Transmit FIFO threshold value (default)
uint8_t tx_curr_thresh;       // Transmit FIFO threshold value (current)
#endif
#if RSCI_CFG_MANC_INCLUDED
uint8_t rx_decoding_pol;      // Decoding conversion select
uint8_t rx_preface_length;    // RX Preface length
uint8_t rx_preface_pattern;   // RX Preface pattern
uint8_t tx_encoding_pol;      // Encoding conversion select
uint8_t tx_preface_length;    // TX Preface length
uint8_t tx_preface_pattern;   // TX Preface pattern
#endif
} rsci_ch_ctrl_t;
```

2.11 Return Values

This section describes return values of API functions. This enumeration is located in r_rsci_rx_if.h as are the prototype declarations of API functions.

```
typedef enum e_rsci_err      // RSCI API error codes
{
    RSCI_SUCCESS=0,
    RSCI_ERR_BAD_CHAN,        // Non-existent channel number
    RSCI_ERR_OMITTED_CHAN,    // RSCI_CHx_INCLUDED is 0 in config.h
    RSCI_ERR_CH_NOT_CLOSED,   // Channel still running in another mode
    RSCI_ERR_BAD_MODE,        // Unsupported or incorrect mode for channel
    RSCI_ERR_INVALID_ARG,     // Argument is not valid for parameter
    RSCI_ERR_NULL_PTR,        // Received null ptr; missing required argument
    RSCI_ERR_XCVR_BUSY,       // Cannot start data transfer; transceiver busy

    // Asynchronous or Manchester
    RSCI_ERR_QUEUE_UNAVAILABLE,  // Cannot open tx or rx queue or both
    RSCI_ERR_INSUFFICIENT_SPACE, // Not enough space in transmit queue
    RSCI_ERR_INSUFFICIENT_DATA,  // Not enough data in receive queue

    // Synchronous/SSPI modes only
    RSCI_ERR_XFER_NOT_DONE    // Data transfer still in progress
} rsci_err_t;
```

2.12 Callback Function

In this module, the callback function specified by the user is called when the RXIn, ERIn interrupt occurs.

The callback function is specified by storing the address of the user function in the “void (* const p_callback)(void *p_args)” structure member (see 2.10, Parameters). When the callback function is called, the variable which stores the constant is passed as the argument.

The argument is passed as void type. Thus the argument of the callback function is cast to a void pointer. See examples below as reference.

When using a value in the callback function, type cast the value.

The following shows an example template for the callback function in asynchronous mode.

```
void MyCallback(void *p_args)
{
rsci_cb_args_t *args;
args = (rsci_cb_args_t *)p_args;
if (args->event == RSCI_EVT_RX_CHAR)
{
//from RXI interrupt; character placed in queue is in args->byte
nop();
}
else if (args->event == RSCI_EVT_RX_CHAR_MATCH)
{
//from RXI interrupt, received data match comparison data
//character placed in queue is in args->byte
nop();
}

#if RSCI_CFG_TEI_INCLUDED
else if (args->event == RSCI_EVT_TEI)
{
// from TEI interrupt; transmitter is idle
// possibly disable external transceiver here
nop();
}
```

```
#endif
else if (args->event == RSCI_EVT_RXBUF_OVFL)
{
// from RXI interrupt; receive queue is full
// unsaved char is in args->byte
// will need to increase buffer size or reduce baud rate
nop();
}
else if (args->event == RSCI_EVT_OVFL_ERR)
{
// from ERI interrupt; receiver overflow error occurred
// error char is in args->byte
// error condition is cleared in ERI routine
nop();
}
else if (args->event == RSCI_EVT_FRAMING_ERR)
{
// from ERI interrupt; receiver framing error occurred
// error char is in args->byte; if = 0, received BREAK condition
// error condition is cleared in ERI routine
nop();
}
else if (args->event == RSCI_EVT_PARITY_ERR)
{
// from ERI interrupt; receiver parity error occurred
// error char is in args->byte
// error condition is cleared in ERI routine
nop();
}
}
```

The following shows an example template for the callback function in SSPI mode.

```
void sspiCallback(void *p_args)
{
rsci_cb_args_t *args;
args = (rsci_cb_args_t *)p_args;
if (args->event == RSCI_EVT_XFER_DONE)
{
// data transfer completed
nop();
}
else if (args->event == RSCI_EVT_XFER_ABORTED)
{
// data transfer aborted
nop();
}
else if (args->event == RSCI_EVT_OVFL_ERR)
{
// from ERI interrupt; receiver overflow error occurred
// error char is in args->byte
// error condition is cleared in ERI interrupt routine
nop();
}
}
```

The following shows an example template for the callback function in manchester mode.

```
void MancCallback(void *p_args)
{
rsci_cb_args_t *args;
```

```
args = (rsci_cb_args_t *)p_args;
if (args->event == RSCI_EVT_RX_CHAR)
{
//from RXI interrupt; character placed in queue is in args->byte
nop();
}
#if RSCI_CFG_TEI_INCLUDED
else if (args->event == RSCI_EVT_TEI)
{
// from TEI interrupt; transmitter is idle
// possibly disable external transceiver here
nop();
}
#endif
else if (args->event == RSCI_EVT_RXBUF_OVFL)
{
// from RXI interrupt; receive queue is full
// unsaved char is in args->byte
// will need to increase buffer size or reduce baud rate
nop();
}
else if (args->event == RSCI_EVT_OVFL_ERR)
{
// from ERI interrupt; receiver overflow error occurred
// error char is in args->byte
// error condition is cleared in ERI routine
nop();
}
else if (args->event == RSCI_EVT_FRAMING_ERR)
{
// from ERI interrupt; receiver framing error occurred
// error char is in args->byte; if = 0, received BREAK condition
// error condition is cleared in ERI routine
nop();
}
else if (args->event == RSCI_EVT_PARITY_ERR)
{
// from ERI interrupt; receiver parity error occurred
// error char is in args->byte
// error condition is cleared in ERI routine
nop();
}
else if (args->event == RSCI_EVT_MANCHESTER_CODE_ERR)
{
// from ERI interrupt; Manchester code error occurred
// error char is in args->byte
// error condition is cleared in ERI routine
nop();
}
else if (args->event == RSCI_EVT_RECEIVE_SYNC_ERR)
{
// from ERI interrupt; receive sync error occurred
// error char is in args->byte;
// error condition is cleared in ERI routine
nop();
}
else if (args->event == RSCI_EVT_START_BIT_ERR)
{
// from ERI interrupt; start bit error occurred
// error char is in args->byte
// error condition is cleared in ERI routine
```

```
nop();
}
else if (args->event == RSCI_EVT_PREFACE_ERR)
{
// from ERI interrupt; preface error occurred
// error char is in args->byte
// error condition is cleared in ERI routine
nop();
}
}
```

This FIT module calls the callback function specified by the user when a receive error interrupt occurs, when 1-byte data is received in asynchronous or manchester mode, when transmissions/receptions for the specified number of bytes have been completed in clock synchronous or SSPI mode, and when a transmit end interrupt occurs.

Note that if the FIFO function is enabled in asynchronous mode, the callback function is executed when receptions for the maximum number of times specified with RSCI_CFG_CHn_RX_FIFO_THRESH have been completed or 15 etu (1) has elapsed from the stop bit of the last received data.

The callback function is set by specifying the address of the callback function to the fourth parameter of R_RSCI_Open(). When the callback function is called, the following parameters are set.

```
typedef struct st_rsci_cb_args      // Arguments of the callback function
{
rsci_hdl_t hdl;          // Handle upon an event occurrence
rsci_cb_evt_t event;     // Event which triggered the event occurred
uint8_t byte;            // Receive data upon an event occurrence
uint8_t num;                   // Receive data size (valid only when FIFO is
used)
} rsci_cb_args_t;

typedef enum e_rsci_cb_evt          // Event for the callback function
{
/* Async/Manc Events */
RSCI_EVT_TEI,             // TEI interrupt occurred; transmitter is idle
RSCI_EVT_RX_CHAR,         // received a character; already placed in queue
RSCI_EVT_RXBUF_OVFL,      // rx queue is full; can't save anymore data
RSCI_EVT_FRAMING_ERR,     // receiver hardware framing error
RSCI_EVT_PARITY_ERR,      // receiver hardware parity error

/* Async Events */
RSCI_EVT_RX_CHAR_MATCH,   // received a matched character; already placed in
queue

/* SSPI/Sync Events */
RSCI_EVT_XFER_DONE,       // transfer completed
RSCI_EVT_XFER_ABORTED,    // transfer aborted

/* Manc Events */
RSCI_EVT_MANCHESTER_CODE_ERR,    // receiver hardware manchester code error
RSCI_EVT_RECEIVE_SYNC_ERR,       // receiver hardware receive sync error
RSCI_EVT_START_BIT_ERR,          // receiver hardware start bit error
RSCI_EVT_PREFACE_ERR,            // receiver hardware preface error

/* Common Events */
RSCI_EVT_OVFL_ERR         // receiver hardware overrun error
} rsci_cb_evt_t;
```

Since the argument is passed as a void pointer, arguments of the callback function must be the pointer variable of type void, for example, when using the argument value within the callback function, it must be type-casted.

Note 1. etu (Elementary Time Unit): 1-bit transfer period

When the following events occur, a received data stored in the argument of the callback function becomes undefined value:

- RSCI_EVT_TEI
- RSCI_EVT_XFER_DONE
- RSCI_EVT_XFER_ABORTED
- RSCI_EVT_OVFL_ERR (when FIFO function enabled)
- RSCI_EVT_PARITY_ERR (when FIFO function enabled)
- RSCI_EVT_FRAMING_ERR (when FIFO function enabled)
- RSCI_EVT_MANCHESTER_CODE_ERR (when Manchester mode is used)
- RSCI_EVT_RECEIVE_SYNC_ERR (when Manchester mode is used)
- RSCI_EVT_START_BIT_ERR (when Manchester mode is used)
- RSCI_EVT_PREFACE_ERR (when Manchester mode is used)

2.13 Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends the method using the Smart Configurator described in (1) or (2) below. However, the Smart Configurator only supports some RX devices. Please use the methods of (3) for RX devices that are not supported by the Smart Configurator.

(1) Adding the FIT module to your project using the Smart Configurator in e^2 studio
By using the Smart Configurator in e^2 studio, the FIT module is automatically added to your project. Refer to “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)” for details.

(2) Adding the FIT module to your project using the Smart Configurator in CS+
By using the Smart Configurator Standalone version in CS+, the FIT module is automatically added to your project. Refer to “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)” for details.

(3) Adding the FIT module to your project in CS+
In CS+, please manually add the FIT module to your project. Refer to “Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)” for details.

2.14 “for”, “while” and “do while” statements

In this module, “for”, “while” and “do while” statements (loop processing) are used in processing to wait for register to be reflected and so on. For these loop processing, comments with “WAIT_LOOP” as a keyword are described. Therefore, if user incorporates fail-safe processing into loop processing, user can search the corresponding processing with “WAIT_LOOP”.

The following shows example of description.

```
while statement example :
/* WAIT_LOOP */
while(0 == SYSTEM.OSCOVFSR.BIT.PLOVF)
{
    /* The delay period needed is to make sure that the PLL has stabilized. */
}

for statement example :
/* Initialize reference counters to 0. */
/* WAIT_LOOP */
for (i = 0; i < BSP_REG_PROTECT_TOTAL_ITEMS; i++)
{
    g_protect_counters[i] = 0;
}

do while statement example :
/* Reset completion waiting */
do
{
    reg = phy_read(ether_channel, PHY_REG_CONTROL);
    count++;
} while ((reg & PHY_CONTROL_RESET) && (count < ETHER_CFG_PHY_DELAY_RESET)); /* WAIT_LOOP */
```

3. API Functions

R_RSCI_Open()

This function applies power to the RSCI channel, initializes the associated registers, enables interrupts, and provides the channel handle for use with other API functions. This function must be called before calling any other API functions.

Format

```
rsci_err_t      R_RSCI_Open (
                uint8_t  const        chan,
                      rsci_mode_t  const   mode,
                rsci_cfg_t * const     p_cfg,
                void                  (* const p_callback)(void *p_args),
                rsci_hdl_t * const    p_hdl
)
```

Parameters

uint8_t const chan
Channel to initialize.

rsci_mode_t const mode
Operational mode (see enumeration below)

*rsci_cfg_t * const p_cfg*
Pointer to configuration union, structure elements (see below) are specific to mode

p_callback
Pointer to function called from interrupt when an RXI or receiver error is detected or for transmit end (TEI) condition
Refer to 2.12, Callback Function for details.

*rsci_hdl_t * const p_hdl*
Pointer to a handle for channel (value set here)
Confirm the return value from R_RSCI_Open is “RSCI_SUCCESS” and then set the first parameter for the other APIs except R_RSCI_GetVersion(). Refer to 2.10, Parameters.

The following RSCI modes are currently supported by this driver module. The mode specified determines the union structure element used for the p_cfg parameter.

```
typedef enum e_rsci_mode     // RSCI operational modes
{
    RSCI_MODE_OFF=0,          // channel not in use
    RSCI_MODE_ASYNC,          // Asynchronous
    RSCI_MODE_SSPI,           // Simple SPI
    RSCI_MODE_SYNC,           // Synchronous
    RSCI_MODE_MANC,           // Manchester
    RSCI_MODE_MAX             // End of modes currently supported
} rsci_mode_t;
```

#defines shown on the next page indicate configurable options for Asynchronous mode used in its configuration structure. These values correspond to bit definitions in the SRC1 and SCR3 registers specify the data length, the parity function, and the STOP bit. The SCR3.CKE and the SCR2.BBR are set using the clock source (8x/16x of the internal/external clock) specified with clk_src of the rsci_uart_t structure and the bit rate specified with baud_rate of the rsci_uart_t structure. Please note this does not guarantee the specified bit rate (there may be some errors depending on the setting). In addition, when using the channel 10 and 11 in the Synchronous mode or SSPI mode with the FIFO feature, you will not be able to set high-speed bit rate than PCLKA/8. (For example, if PCLKA is 120 MHz, it is possible to set the bit rate of equal to or less than 15 Mbps.)

The following shows the union for p_cfg:

```
typedef union
{
    rsci_uart_t      async;
    rsci_sync_sspi_t sync;
    rsci_sync_sspi_t sspi;
    rsci_manc_t      manc;
} rsci_cfg_t;
```

The following shows the structure used for settings in Asynchronous mode:

```
typedef struct st_rsci_uart
{
    uint32_t    baud_rate;      // ie 9600, 19200, 115200
    uint8_t     clk_src;        // use RSCI_CLK_INT/EXT8X/EXT16X
    uint8_t     data_size;      // use RSCI_DATA_nBIT
    uint8_t     parity_type;    // use RSCI_ODD/EVEN/NONE_PARITY
    uint8_t     stop_bits;      // use RSCI_STOPBITS_1/2
    bool        msb_first;
    bool        invert_data;
    uint8_t     int_priority;   // interrupt priority; 1=low, 15=high
} rsci_uart_t;
```

The following shows the definitions of the structure (rsci_uart_t) members used in Asynchronous mode:

```
/* Definitions for the sck_src member. */
#define RSCI_CLK_INT         0x00 // use internal clock for baud rate generation
#define RSCI_CLK_EXT_8X      0x03  // use external clock 8x baud rate
#define RSCI_CLK_EXT_16X     0x02  // use external clock 16x baud rate

/* Definitions for the data_size member. */
#define RSCI_DATA_7BIT       0x30   // 7-bit length (LSB is fixed)
#define RSCI_DATA_8BIT       0x20   // 8-bit length

/* Definitions for the parity_type member. */
#define RSCI_ODD_PARITY      0x01   // Odd parity
#define RSCI_EVEN_PARITY     0x00   // Even parity
#define RSCI_NONE_PARITY     0x02   // No parity

/* Definitions for the stop_bits member.
#define RSCI_STOPBITS_2      0x01   // 2-stop bit
#define RSCI_STOPBITS_1      0x00   // 1-stop bit
```

The following shows the structure used for settings in SSPI and Synchronous modes:

```
typedef struct st_rsci_sync_sspi
{
    rsci_spi_mode_t  spi_mode;      // clock polarity and phase; unused for sync
    uint32_t        bit_rate;      // ie 1000000 for 1Mbps
    bool            msb_first;
    bool            invert_data;
    uint8_t         int_priority;   // rxi,eri interrupt priority; 1=low,
15=high
```

```
} rsci_sync_sspi_t;
```

The following shows the enumeration used for spi_mode of the rsci_sync_sspi_t structure in SSPI or Synchronous mode:

```
typedef enum e_rsci_spi_mode
{
    RSCI_SPI_MODE_OFF = 4,   /* channel is in synchronous mode */
    RSCI_SPI_MODE_0 = 0x00,  /* SCR3 Register CPHA=0, CPOL=0; Mode 0: 00 */
    RSCI_SPI_MODE_1 = 0x01,  /* SCR3 Register CPHA=1, CPOL=0; Mode 1: 01 */
    RSCI_SPI_MODE_2 = 0x02,  /* SCR3 Register CPHA=0, CPOL=1; Mode 2: 10 */
    RSCI_SPI_MODE_3 = 0x03   /* SCR3 Register CPHA=1, CPOL=1; Mode 3: 11 */
} rsci_spi_mode_t;
```

The following shows the structure used for settings in Manchester mode:

```
typedef struct st_rsci_manc
{
    uint32_t    baud_rate;      // ie 9600, 19200, 115200
    uint8_t     data_size;      // use RSCI_DATA_nBIT
    uint8_t     parity_type;    // use RSCI_ODD/EVEN/NONE_PARITY
    uint8_t     start_bits;     // use RSCI_STARTBITS_1/3
    uint8_t     stop_bits;      // use RSCI_STOPBITS_1/2
    bool        msb_first;
    bool        invert_data;
    uint8_t     int_priority;   // interrupt priority; 1=low, 15=high
} rsci_manc_t;
```

The following shows the definitions of the structure (rsci_manc_t) members used in Manchester mode:

```
/* Definitions for the data_size member. */
#define RSCI_DATA_7BIT       0x30   // 7-bit length (LSB is fixed)
#define RSCI_DATA_8BIT       0x20   // 8-bit length

/* Definitions for the parity_type member. */
#define RSCI_ODD_PARITY      0x01   // Odd parity
#define RSCI_EVEN_PARITY     0x00   // Even parity
#define RSCI_NONE_PARITY     0x02   // No parity

/* Definitions for the stop_bits member.
#define RSCI_STOPBITS_2      0x01   // 2-stop bit
#define RSCI_STOPBITS_1      0x00   // 1-stop bit

/* Definitions for the start_bits member.
#define RSCI_STARTBITS_3      0x01  // 3-start bit
#define RSCI_STARTBITS_1      0x00  // 1-start bit
```

Return Values

[RSCI_SUCCESS]	*/* Successful; channel initialized */*
[RSCI_ERR_BAD_CHAN]	*/* Channel number is invalid for part*/*
[RSCI_ERR_OMITTED_CHAN]	*/* Corresponding RSCI_CHx_INCLUDED is invalid (0) */*
[RSCI_ERR_CH_NOT_CLOSED]	*/* Channel currently in operation; Perform R_RSCI_Close() first*/*
[RSCI_ERR_BAD_MODE]	*/* Mode specified not currently supported*/*
[RSCI_ERR_NULL_PTR]	*/* p_cfg pointer is NULL*/*
[RSCI_ERR_INVALID_ARG]	*/* An element of the p_cfg structure contains an invalid value. */*
[RSCI_ERR_QUEUE_UNAVAILABLE]	*/* Cannot open transmit or receive queue or both (Asynchronous or Manchester mode) */*

Properties

Prototyped in file “r_rsci_rx_if.h”

Description

Initializes an RSCI channel for a particular mode and provides a Handle in **p_hdl* for use with other API functions. RXI and ERI interrupts are enabled in all modes. TXI is enabled in Asynchronous or Manchester mode.

Example: Asynchronous Mode

```
    rsci_cfg_t   config;
    rsci_hdl_t   Console;
    rsci_err_t   err;

    config.async.baud_rate = 115200;
    config.async.clk_src = RSCI_CLK_INT;
    config.async.data_size = RSCI_DATA_8BIT;
    config.async.parity_type = RSCI_NONE_PARITY;
    config.async.stop_bits = RSCI_STOPBITS_1;
    config.async.msb_first = false;
    config.async.data_invert = false;
    config.async.int_priority = 2;                // 1=lowest, 15=highest

    err = R_RSCI_Open(RSCI_CH10, RSCI_MODE_ASYNC, &config, MyCallback,
&Console);
```

Example: SSPI Mode

```
    rsci_cfg_t   config;
    rsci_hdl_t   sspiHandle;
    rsci_err_t   err;

    config.sspi.spi_mode = RSCI_SPI_MODE_0;
    config.sspi.bit_rate = 1000000;           // 1 Mbps
    config.sspi.msb_first = true;
    config.sspi.invert_data = false;
    config.sspi.int_priority = 4;
    err = R_RSCI_Open(RSCI_CH10, RSCI_MODE_SSPI, &config, sspiCallback,
&sspiHandle);
```

Example: Synchronous Mode

```
    rsci_cfg_t   config;
    rsci_hdl_t   syncHandle;
    rsci_err_t   err;

    config.sync.spi_mode = RSCI_SPI_MODE_OFF;
    config.sync.bit_rate = 1000000;           // 1 Mbps
    config.sync.msb_first = true;
    config.sync.invert_data = false;
    config.sync.int_priority = 4;
    err = R_RSCI_Open(RSCI_CH10, RSCI_MODE_SYNC, &config, syncCallback,
&syncHandle);
```

Example: Manchester Mode

```
    rsci_cfg_t   config;
    rsci_hdl_t   Console;
    rsci_err_t   err;

    config.manc.baud_rate = 115200;
    config.manc.data_size = RSCI_DATA_8BIT;
    config.manc.parity_type = RSCI_NONE_PARITY;
    config.manc.stop_bits = RSCI_STOPBITS_1;
```

```
    config.manc.stop_bits = RSCI_STARTBITS_1;
    config.manc.msb_first = false;
    config.manc.data_invert = false;
    config.manc.int_priority = 2;                // 1=lowest, 15=highest

    err = R_RSCI_Open(RSCI_CH10, RSCI_MODE_MANC, &config, ManCallback,
&Console);
```

Special Notes:

The driver calculates the optimum values for SCR2.BRR, SCR2.ABCS, and SCR2.CKS using BSP_PCLKA_HZ and BSP_PCLKB_HZ as defined in mcu_info.h of the board support package. This however does not guarantee a low bit error rate for all peripheral clock/baud rate combinations.

If an external clock is used in Asynchronous mode, the pin direction must be selected before calling the R_RSCI_Open() function, and the pin function and mode must be selected after calling the R_RSCI_Open() function. The following is an example initialization for RX671 channel 10:

```
Before the R_RSCI_Open() function call

    PORT8.PDR.BIT.B0 = 0;       // set SCK010 pin direction to input (dflt)

After the R_RSCI_Open() function call

    MPC.P80PFS.BYTE = 0x2C;     // Pin Func Select P80 SCK010
    PORT8.PMR.BIT.B0 = 1;       // set SCK pin mode to peripheral
```

For settings of the pins used for communications, the pin directions and their outputs must be selected before calling the R_RSCI_Open() function, and the pin functions and modes must be selected after calling the R_RSCI_Open() function.
An example for initializing channel 10 for SSPI on the RX671 is as follows:

```
Before the R_RSCI_Open() function call

    PORT8.PODR.BIT.B2 = 0;      // set line low
    PORT8.PODR.BIT.B1 = 0;      // set line low
    PORT8.PDR.BIT.B0 = 1;       // set clock pin direction to output
    PORT8.PDR.BIT.B2 = 1;       // set MOSI pin direction to output
    PORT8.PDR.BIT.B1 = 0;       // set MISO pin direction to input

After the R_RSCI_Open() function call

    MPC.P82PFS.BYTE = 0x2C;     // Pin Func Select P82 MOSI
    MPC.P81PFS.BYTE = 0x2C;     // Pin Func Select P81 MISO
    MPC.P80PFS.BYTE = 0x2C;     // Pin Func Select P80 SCK010
    PORT8.PMR.BIT.B2 = 1;       // set MOSI pin mode to peripheral
    PORT8.PMR.BIT.B1 = 1;       // set MISO pin mode to peripheral
    PORT8.PMR.BIT.B0 = 1;       // set clock pin mode to peripheral
```

When using Asynchronous or Manchester mode, two bytes queues are used for one channel. Adjust the number of byte queues as necessary. Refer to the application note "BYTEQ Module Using Firmware Integration Technology (R01AN1683)" for details.

R_RSCI_Close()

This function removes power from the RSCI channel and disables the associated interrupts.

Format

rsci_err_t R_RSCI_Close (

rsci_hdl_t const hdl

)

Parameters

rsci_hdl_t const hdl

Handle for channel

Set *hdl* when R_RSCI_Open() is successfully processed.

Return Values

[RSCI_SUCCESS] / Successful; channel closed */*

[RSCI_ERR_NULL_PTR] / hdl is NULL */*

Properties

Prototyped in file "r_rsci_rx_if.h"

Description

Disables the RSCI channel designated by the handle and enters module-stop state.

Example

```
rsci_hdl_t   Console;
  ...
err = R_RSCI_Open(RSCI_CH10, RSCI_MODE_ASYNC, &config, MyCallback, &Console);
  ...
err = R_RSCI_Close(Console);
```

Special Notes:

This function will abort any transmission or reception that may be in progress.

R_RSCI_Send()

Initiates transmit if transmitter is not in use. Queues data for later transmit when in Asynchronous or Manchester mode.

Format

```
rsci_err_t      R_RSCI_Send (
                    rsci_hdl_t const    hdl,
                    uint8_t             *p_src,
                    uint16_t const      length
)
```

Parameters

rsci_hdl_t const hdl
Handle for channel
Set *hdl* when R_RSCI_Open() is successfully processed.

uint8_t p_src*
Pointer to data to transmit

uint16_t const length
Number of bytes to send

Return Values

[RSCI_SUCCESS]	*/* Transmit initiated or loaded into queue (Asynchronous/Manchester) */*
[RSCI_ERR_NULL_PTR]	*/* hdl value is NULL */*
[RSCI_ERR_BAD_MODE]	*/* Mode specified not currently supported */*
[RSCI_ERR_INSUFFICIENT_SPACE]	*/* Insufficient space in queue to load all data (Asynchronous/Manchester) */*
[RSCI_ERR_XCVR_BUSY]	*/* Channel currently busy (SSPI/Synchronous) */*

Properties

Prototyped in file “r_rsci_rx_if.h”

Description

In asynchronous or manchester mode, this function places data into a transmit queue if the transmitter for the RSCI channel referenced by the handle is not in use. In SSPI and Synchronous modes, no data is queued and transmission begins immediately if the transceiver is not already in use.

Note that the toggling of Slave Select lines when in SSPI mode is not handled by this driver. The Slave Select line for the target device must be enabled prior to calling this function.

Also, toggling of the CTS/RTS pin in Synchronous/Asynchronous/Manchester mode is not handled by this driver.

Example: Asynchronous/Manchester Mode

```
#define STR_CMD_PROMPT "Enter Command: "
rsci_hdl_t Console;
rsci_err_t err;

err = R_RSCI_Send(Console, STR_CMD_PROMPT, sizeof(STR_CMD_PROMPT));
```

```
    // Cannot block for this transfer to complete. However, can use TEI
interrupt
    // to determine when there is no more data in queue left to transmit.
```

Example: SSPI Mode

```
    rsci_hdl_t  sspiHandle;
    rsci_err_t  err;
    uint8_t     flash_cmd,sspi_buf[10];

    // SEND COMMAND TO FLASH DEVICE TO PROVIDE ID */
    FLASH_SS = SS_ON;                // enable gpio flash slave select
    flash_cmd = SF_CMD_READ_ID;

    R_RSCI_Send(sspiHandle, &flash_cmd, 1);
    while (RSCI_SUCCESS != R_SCI_Control(sspiHandle, SCI_CMD_CHECK_XFER_DONE,
NULL))
    {
    }

    /* READ ID FROM FLASH DEVICE */
    R_RSCI_Receive(sspiHandle, sspi_buf, 5);
    while (RSCI_SUCCESS != R_RSCI_Control(sspiHandle, RSCI_CMD_CHECK_XFER_DONE,
NULL))
    {
    }

    FLASH_SS = SS_OFF;               // disable gpio flash slave select
```

Example: Synchronous Mode

```
    #define STRING1 "Test String"
    rsci_hdl_t  lcdHandle;
    rsci_err_t  err;


    // SEND STRING TO LCD DISPLAY AND WAIT TO COMPLETE */
    R_RSCI_Send(lcdHandle, STRING1, sizeof(STRING1));

    while (RSCI_SUCCESS != R_RSCI_Control(lcdHandle, RSCI_CMD_CHECK_XFER_DONE,
NULL))
    {
    }
```

Special Notes:

None.

R_RSCI_Receive()

In Asynchronous or Manchester mode, fetches data from a queue which is filled by RXI interrupts. In other modes, initiates reception if transceiver is not in use.

Format

```
rsci_err_t      R_RSCI_Receive (
                rsci_hdl_t const    hdl,
                uint8_t             *p_dst,
                uint16_t const      length
)
```

Parameters

rsci_hdl_t const hdl
Handle for channel
Set *hdl* when R_RSCI_Open() is successfully processed.

uint8_t p_dst*
Pointer to buffer to load data into

uint16_t const length
Number of bytes to read

Return Values

[RSCI_SUCCESS]	*/* Requested number of bytes were loaded into p_dst (Asynchronous/Manchester) Clocking in of data initiated (SSPI/Synchronous)*
[RSCI_ERR_NULL_PTR]	*/* hdl value is NULL*
[RSCI_ERR_BAD_MODE]	*/* Mode specified not currently supported*
[RSCI_ERR_INSUFFICIENT_DATA]	*/* Insufficient data in receive queue to fetch all data (Asynchronous/Manchester)*
[RSCI_ERR_XCVR_BUSY]	*/* Channel currently busy (SSPI/Synchronous)*

Properties

Prototyped in file “r_rsci_rx_if.h”

Description

In Asynchronous or Manchester mode, this function gets data received on an RSCI channel referenced by the handle from its receive queue. This function will not block if the requested number of bytes is not available. In SSPI/Synchronous modes, the clocking in of data begins immediately if the transceiver is not already in use. The value assigned to RSCI_CFG_DUMMY_TX_BYTE in r_rsci_config.h is clocked out while the receive data is being clocked in.

If any errors occurred during reception, the callback function specified in R_RSCI_Open() is executed. Check an event passed with the argument of the callback function to see if the reception has been successfully completed. Refer to 2.12, Callback Function for details.

Note that the toggling of Slave Select lines when in SSPI mode is not handled by this driver. The Slave Select line for the target device must be enabled prior to calling this function.

Example: Asynchronous/Manchester Mode

```
rsci_hdl_t Console;
rsci_err_t err;
```

```
    uint8_t   byte;

    /* echo characters */
    while (1)
    {
        while (RSCI_SUCCESS != R_RSCI_Receive(Console, &byte, 1))
        {
        }
        R_RSCI_Send(Console, &byte, 1);
    }
```

Example: SSPI Mode

```
    rsci_hdl_t  sspiHandle;
    rsci_err_t  err;
    uint8_t     flash_cmd,sspi_buf[10];

    // SEND COMMAND TO FLASH DEVICE TO PROVIDE ID */

    FLASH_SS = SS_ON;               // enable gpio flash slave select
    flash_cmd = SF_CMD_READ_ID;

    R_RSCI_Send(sspiHandle, &flash_cmd, 1);
    while (RSCI_SUCCESS != R_RSCI_Control(sspiHandle, RSCI_CMD_CHECK_XFER_DONE,
NULL))
    {
    }

    /* READ ID FROM FLASH DEVICE */
    R_RSCI_Receive(sspiHandle, sspi_buf, 5);
    while (RSCI_SUCCESS != R_RSCI_Control(sspiHandle, RSCI_CMD_CHECK_XFER_DONE,
NULL))
    {
    }

    FLASH_SS = SS_OFF;              // disable gpio flash slave select
```

Example: Synchronous Mode

```
    rsci_hdl_t  sensorHandle;
    rsci_err_t  err;
    uint8_t     sensor_cmd,sync_buf[10];

    // SEND COMMAND TO SENSOR TO PROVIDE CURRENT READING */

    sensor_cmd = SNS_CMD_READ_LEVEL;

    R_RSCI_Send(sensorHandle, &sensor_cmd, 1);
    while (RSCI_SUCCESS != R_RSCI_Control(sensorHandle,
RSCI_CMD_CHECK_XFER_DONE, NULL))
    {
    }

    /* READ LEVEL FROM SENSOR */
    R_RSCI_Receive(sensorHandle, sync_buf, 4);
    while (RSCI_SUCCESS != R_RSCI_Control(sensorHandle,
RSCI_CMD_CHECK_XFER_DONE, NULL))
    {
    }
```

Special Notes:

See section 2.12 Callback Function for values passed to arguments of the callback function.

In Asynchronous mode, when data match detected, received data stored in a queue and notify to user by callback function with event RSCI_EVT_RX_CHAR_MATCH.

R_RSCI_SendReceive()

For Synchronous and SSPI modes only. Transmits and receives data simultaneously if the transceiver is not in use.

Format

```
rsci_err_t      R_SCI_SendReceive (
                rsci_hdl_t const   hdl,
                uint8_t            *p_src,
                uint8_t            *p_dst,
                uint16_t const     length
)
```

Parameters

rsci_hdl_t const hdl
Handle for channel
Set *hdl* when R_RSCI_Open() is successfully processed.

uint8_t p_src*
Pointer to data to transmit

uint8_t p_dst*
Pointer to buffer to load data into

uint16_t const length
Number of bytes to send

Return Values

[RSCI_SUCCESS] *\/* Data transfer initiated *\/*
[RSCI_ERR_NULL_PTR] *\/* hdl value is NULL *\/*
[RSCI_ERR_BAD_MODE] *\/* Channel mode not SSPI or Synchronous *\/*
[RSCI_ERR_XCVR_BUSY] *\/* Channel currently busy *\/*

Properties

Prototyped in file “r_rsci_rx_if.h”

Description

If the transceiver is not in use, this function clocks out data from the p_src buffer while simultaneously clocking in data and placing it in the p_dst buffer.

Note that the toggling of Slave Select lines for SSPI is not handled by this driver. The Slave Select line for the target device must be enabled prior to calling this function.

Also, toggling of the CTS/RTS pin in Synchronous/Asynchronous mode is not handled by this driver.

Example: SSPI Mode

```
    rsci_hdl_t  sspiHandle;
    rsci_err_t  err;
    uint8_t in_buf[2] = {0x55, 0x55};     // init to illegal values


    /* READ FLASH STATUS USING SINGLE API CALL */

    // load array with command to send plus one dummy byte for clocking in
status reply
```

```
    uint8_t out_buf[2] = {SF_CMD_READ_STATUS_REG, RSCI_CFG_DUMMY_TX_BYTE };

    FLASH_SS = SS_ON;

    err = R_RSCI_SendReceive(sspiHandle, out_buf, in_buf, 2);
    while (RSCI_SUCCESS != R_RSCI_Control(sspiHandle, RSCI_CMD_CHECK_XFER_DONE,
NULL))
    {
    }

    FLASH_SS = SS_OFF;

    // in_buf[1] contains status
```

Special Notes:

See section 2.12 Callback Function for values passed to arguments of the callback function.

R_RSCI_Control()

This function configures and controls the operating mode for the RSCI channel.

Format

```
rsci_err_t      R_RSCI_Control (
                rsci_hdl_t const     hdl,
                rsci_cmd_t const     cmd,
                void                 *p_args
)
```

Parameters

rsci_hdl_t const hdl

Handle for channel

Set *hdl* when R_RSCI_Open() is successfully processed.

rsci_cmd_t const cmd

Command to run (see enumeration below)

*void *p_args*

Pointer to arguments (see below) specific to command, casted to void *

The valid *cmd* values are as follows:

```
typedef enum e_rsci_cmd          // RSCI Control() commands
{
    /* All modes */
    RSCI_CMD_CHANGE_BAUD,                /* change baud/bit rate */
#if ((RSCI_CFG_CH10_FIFO_INCLUDED) || (RSCI_CFG_CH11_FIFO_INCLUDED))
    RSCI_CMD_CHANGE_TX_FIFO_THRESH,      /* change TX FIFO threshold */
    RSCI_CMD_CHANGE_RX_FIFO_THRESH,      /* change RX FIFO threshold */
#endif
#if defined(BSP_MCU_RX671)
    RSCI_CMD_SET_RXI_PRIORITY,           /* change RXI priority level */
    RSCI_CMD_SET_TXI_PRIORITY,           /* change TXI priority level */
#endif
    RSCI_CMD_XFER_LSB_FIRST,             /* start from LSB bit when sending */
    RSCI_CMD_XFER_MSB_FIRST,             /* start from MSB bit when sending */
    RSCI_CMD_INVERT_DATA,                /* logic level of send/receive data is
invert */

    /* Async commands */
    RSCI_CMD_EN_NOISE_CANCEL,            /* enable noise cancellation */
    RSCI_CMD_EN_TEI,                     /* RSCI_CMD_EN_TEI is obsolete command,
                                           but it exists only for compatibility

with older version. */
    RSCI_CMD_OUTPUT_BAUD_CLK,            /* output baud clock on the SCK pin */
    RSCI_CMD_START_BIT_EDGE,             /* detect start bit as falling edge of
RXDn pin

                                          (default detect as low level on RXDn
pin) */
    RSCI_CMD_GENERATE_BREAK,             /* generate break condition */
    RSCI_CMD_COMPARE_RECEIVED_DATA,      /* Compare received data with comparison
data */

    /* Async/Manc commands */
    RSCI_CMD_TX_Q_FLUSH,                 /* flush transmit queue */
    RSCI_CMD_RX_Q_FLUSH,                 /* flush receive queue */
```

```
    RSCI_CMD_TX_Q_BYTES_FREE,          /* get count of unused transmit queue
bytes */
    RSCI_CMD_RX_Q_BYTES_AVAIL_TO_READ, /* get num bytes ready for reading */

    /* Async/Sync/Manc commands*/
    RSCI_CMD_EN_CTS_IN,                /* enable CTS input (default RTS output)
*/

    /* SSPI/Sync commands */
    RSCI_CMD_CHECK_XFER_DONE,          /* see if send, rcv, or both are done;
RSCI_SUCCESS if yes */
    RSCI_CMD_ABORT_XFER,

    /* SSPI commands */
    RSCI_CMD_CHANGE_SPI_MODE,          /* change clock polarity and phase in
SSPI mode */
    RSCI_CMD_CHECK_TX_DONE,            /* see if tx requests complete;
RSCI_SUCCESS if yes */
    RSCI_CMD_CHECK_RX_DONE,            /* see if rx request complete in sync
mode; RSCI_SUCCESS if yes */
    RSCI_CMD_CHECK_RX_SYNC_DONE,

    /*Sampling/transition timing adjust commands*/
    RSCI_CMD_RX_SAMPLING_ENABLE,
    RSCI_CMD_RX_SAMPLING_DISABLE,
    RSCI_CMD_TX_TRANSITION_TIMING_ENABLE,
    RSCI_CMD_TX_TRANSITION_TIMING_DISABLE,
    RSCI_CMD_SAMPLING_TIMING_ADJUST,
    RSCI_CMD_TRANSITION_TIMING_ADJUST,

    /* Manchester commands */
    RSCI_CMD_START_BIT_PATTERN_LOW_TO_HIGH, /* start bit is a low to high */
    RSCI_CMD_START_BIT_PATTERN_HIGH_TO_LOW, /* start bit is a high to low */
    RSCI_CMD_EN_SYNC,                       /* enable sync, start bit pattern
set with SYNC bit in TDR */
    RSCI_CMD_SET_TRASMIT_PREFACE_LENGTH,    /* set preface length in transmit
data */
    RSCI_CMD_SET_RECEIVE_PREFACE_LENGTH     /* set preface length in received
frames */
} rsci_cmd_t;
```

Commands other than the following command do not require arguments and take FIT_NO_PTR for p_args.

The argument for RSCI_CMD_CHANGE_BAUD is a pointer to the rsci_baud_t variable containing the new bit rate desired. The rsci_baud_t structure is shown below.

```
typedef struct st_rsci_baud
{
    uint32_t    pclk;       // peripheral clock speed; e.g. 24000000 is 24 MHz
    uint32_t    rate;       // e.g. 9600, 19200, 115200
} rsci_baud_t;
```

The argument for RSCI_CMD_TX_Q_BYTES_FREE and RSCI_CMD_RX_Q_BYTES_AVAIL_TO_READ is a pointer to a uint16_t variable to hold a count value.

The argument for RSCI_CMD_CHANGE_SPI_MODE is a pointer to the enumeration (rsci_sync_sspi_t) variable containing the new mode desired.

The argument for RSCI_CMD_SET_TXI_PRIORITY and RSCI_CMD_SET_RXI_PRIORITY (for MCU which can specify different priority levels for TXI and RXI) is a pointer to a uint8_t variable to hold the priority level.

Return Values

[RSCI_SUCCESS] */* Successful; channel initialized */*
[RSCI_ERR_NULL_PTR] */* hdl or p_args pointer is NULL (when required) */*
[RSCI_ERR_BAD_MODE] */* Mode specified not currently supported */*
[RSCI_ERR_INVALID_ARG] */* The cmd value or an element of p_args contains an invalid value. */*

Properties

Prototyped in file "r_rsci_rx_if.h"

Description

This function is used for configuring special hardware features such as changing driver configuration and obtaining driver status.

The CTS/ RTS pin functions as RTS by default hardware control. By issuing an RSCI_CMD_EN_CTS_IN, the pin functions as CTS.

Example: Asynchronous Mode

```
    rsci_hdl_t   Console;
    rsci_cfg_t   config;
    rsci_baud_t  baud;
    rsci_err_t   err;
    uint16_t    cnt;

    R_RSCI_Open(RSCI_CH10, RSCI_MODE_ASYNC, &config, MyCallback, &Console);
    R_RSCI_Control(Console, RSCI_CMD_EN_NOISE_CANCEL, NULL);
    R_RSCI_Control(Console, RSCI_CMD_EN_TEI, NULL);
      ...
    /* reset baud rate due to low power mode clock switching */
    baud.pclk = 8000000;      // 8 MHz
    baud.rate = 19200;
    R_RSCI_Control(Console, RSCI_CMD_CHANGE_BAUD, (void *)&baud);
      ...
    /* after sending several messages, determine how much space is left in tx
queue */
    R_RSCI_Control(Console, RSCI_CMD_TX_Q_BYTES_FREE, (void *)&cnt);
      ...
    /* check to see if there is data sitting in the receive queue */
    R_RSCI_Control(Console, RSCI_CMD_RX_Q_BYTES_AVAIL_TO_READ, (void *)&cnt);
```

Example: SSPI Mode

```
    rsci_cfg_t   config;
    rsci_spi_mode_t   mode;
    rsci_hdl_t   sspiHandle;
    rsci_err_t   err;

    config.sspi.spi_mode     = RSCI_SPI_MODE_0;
    config.sspi.bit_rate     = 1000000;           // 1 Mbps
    config.sspi.msb_first    = true;
    config.sspi.invert_data  = false;
    config.sspi.int_priority = 4;
    err = R_RSCI_Open(RSCI_CH10, RSCI_MODE_SSPI, &config, sspiCallback,
&sspiHandle);
      ...
      ...
    // for changing to slave device which operates in a different mode
    mode = RSCI_SPI_MODE_3;
    R_RSCI_Control(sspiHandle, RSCI_CMD_CHANGE_SPI_MODE, (void *)&mode);
```

Example: Manchester Mode

```
    rsci_hdl_t   Console;
    rsci_cfg_t   config;
    rsci_baud_t  baud;
    rsci_err_t   err;
    uint16_t    cnt;

    R_RSCI_Open(RSCI_CH10, RSCI_MODE_MANC, &config, MancCallback, &Console);
    R_RSCI_Control(Console, RSCI_CMD_START_BIT_PATTERN_HIGH_TO_LOW, NULL);
      ...
    /* reset baud rate due to low power mode clock switching */
    baud.pclk = 8000000;      // 8 MHz
    baud.rate = 19200;
    R_RSCI_Control(Console, RSCI_CMD_CHANGE_BAUD, (void *)&baud);
      ...
    /* after sending several messages, determine how much space is left in tx 
queue */
    R_RSCI_Control(Console, RSCI_CMD_TX_Q_BYTES_FREE, (void *)&cnt);
      ...
    /* check to see if there is data sitting in the receive queue */
    R_RSCI_Control(Console, RSCI_CMD_RX_Q_BYTES_AVAIL_TO_READ, (void *)&cnt);
```

Special Notes:

When RSCI_CMD_CHANGE_BAUD is used, the optimum values for SCR2.BRR, SCR2.ABCS, and SCR2.CKS is calculated based on the bit rate specified. This however does not guarantee a low bit error rate for all peripheral clock/baud rate combinations.

If the command RSCI_CMD_EN_CTS_IN is to be used, the pin direction must be selected before calling the R_RSCI_Open() function, and the pin function and mode must be selected after calling the R_RSCI_Open() function. The following is an example initialization for RX671 channel 10:

```
Before the R_RSCI_Open() function call

PORTC.PDR.BIT.B4 = 0;       // set CTS/RTS pin direction to input (dflt)

After the R_RSCI_Open() function call

MPC.PC4PFS.BYTE = 0x2C;     // Pin Func Select PC4 CTS
PORTC.PMR.BIT.B4 = 1;       // set CTS/RTS pin mode to peripheral
```

If the command RSCI_CMD_OUTPUT_BAUD_CLK is to be used, the pin direction must be selected before calling the R_RSCI_Open() function, and the pin function and mode must be selected after calling the R_RSCI_Open() function.
The following is an example initialization for RX671 channel 10:

```
Before the R_RSCI_Open() function call

    PORT8.PDR.BIT.B0 = 1;       // set SCK010 pin direction to output

After the R_RSCI_Open() function call

    MPC.P80PFS.BYTE = 0x2C;     // Pin Func Select P80 SCK010
    PORT8.PMR.BIT.B0 = 1;       // set SCK010 pin mode to peripheral
```

If the command RSCI_CMD_EN_SYNC is to be used, enable it using SYNC(Sync Pulse Select) bit in TDR. If the Start Bit pattern is set with SYNC bit (Data or Command Sync), the value setting of Start Bit Length is automatically changed to 3 bits length.

The commands listed below can be executed during transmission. Do not execute the other commands during transmission.

- RSCI_CMD_TX_Q_BYTES_FREE
- RSCI_CMD_RX_Q_BYTES_AVAIL_TO_READ
- RSCI_CMD_CHECK_XFER_DONE
- RSCI_CMD_ABORT_XFER

When this function is executed, the TXD pin temporarily becomes Hi-Z. Use any of the following methods to prevent the TXD pin from becoming Hi-Z.

<u>When the RSCI_CMD_GENERATE_BREAK command is used:</u>

- Connect the TXD pin to Vcc via a resistor (pull-up).

<u>When a command other than above is used:</u>

Perform one of the following methods:

- Connect the TXD pin to Vcc via a resistor (pull-up).
- Switch the pin function of the TXD pin to general I/O port before the RSCI_Control function is executed. Then switch it back to peripheral function after the RSCI_Control function has been executed.

R_RSCI_GetVersion()

This function returns the driver version number at runtime.

Format

uint32_t R_RSCI_GetVersion (void)

Parameters

None

Return Values

Version number.

Properties

Prototyped in file "r_rsci_rx_if.h"

Description

Returns the version of this module. The version number is encoded such that the top 2 bytes are the major version number and the bottom 2 bytes are the minor version number.

Example

```
uint32_t   version;
  ...
version = R_RSCI_GetVersion();
```

Special Notes:

None.

4. Pin Setting

To use the RSCI FIT module, assign input/output signals of the peripheral function to pins with the multi-function pin controller (MPC). The pin assignment is referred to as the “Pin Setting” in this document.

Please perform the pin setting after calling the R_RSCI_Open function.

When performing the pin setting in the e^2 studio, the Pin Setting feature of the the Smart Configurator can be used. When using the Pin Setting feature, a source file is generated according to the option selected in the Pin Setting window in the Smart Configurator. Then pins are configured by calling the function defined in the source file. Refer to Table 4.1 Function Output by the Smart Configurator for details.

Table 4.1 Function Output by the Smart Configurator

MCU Used	Function to be Output	Remarks
All MCUs	R_RSCI_PinSet_RSCIx	x: Channel number

5. Demo Projects

Demo projects include function main() that utilizes the FIT module and its dependent modules (e.g. r_bsp). This FIT module includes the following demo projects.

5.1 rsci_demo_rskrx671, rsci_demo_rskrx671_gcc

This is a simple demo of the RX671 Serial Communications Interface (RSCI) for the RSKRX671 starter kit (FIT module "r_rsci_rx"). In the demo project, the MCU communicates with the terminal through the RSCI channel configured as the UART. The RS232 interface is not on the RSKRX671 in the demo, thus the USB virtual COM interface is used as serial interface for RSKRX671. A PC running the terminal emulation application is required for communicating with the user.

Setup and Execution

1. Build this sample application, download it to the RSK board, and execute the application using a debugger.
2. Connect the serial port on the RSK board to the serial port on the PC.

   This demo program uses the USB virtual COM interface. In this case, connect the serial port to the USB port on the PC where the Renesas USB serial device driver is installed.
3. Open the terminal emulation program on the PC and select the serial COM port allocated to the USB serial virtual COM interface on the RSK.
4. Configure the terminal serial settings so that they correspond to the settings in this sample application listed below:
   115200 bps, 8-bit data, no parity, 1 stop bit, no flow control
5. The software waits for receiving characters from the terminal.
   When the terminal program on the PC is ready, press a key on the keyboard in the PC's terminal window and check the version number of the FIT module output on the terminal.
6. This application is in echo mode. A given key input to the terminal is received by the RSCI driver and then the application returns the characters to the terminal.

Boards Supported

RSKRX671

6. Appendices

6.1 Confirmed Operation Environment

This section describes confirmed operation environment for the RSCI FIT module.

Table 6.1 Confirmed Operation Environment (Rev.1.00)

Item	Contents
Integrated development environment	Renesas Electronics e² studio Version 21.7.0 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.03.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202004 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.1.00
Board used	Renesas Starter Kit+ for RX671 (product No.: RTK55671xxxxxxxxxx)

Table 6.2 Confirmed Operation Environment (Rev.1.10)

Item	Contents
Integrated development environment	Renesas Electronics e² studio Version 21.7.0 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.03.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202004 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.1.10
Board used	Renesas Starter Kit+ for RX671 (product No.: RTK55671xxxxxxxxxx)

Table 6.3 Confirmed Operation Environment (Rev.2.00)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 21.10.0 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.03.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202102 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.00
Board used	Renesas Starter Kit+ for RX671 (product No.: RTK55671xxxxxxxxxx)

Table 6.4 Confirmed Operation Environment (Rev.2.10)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 22.4.0 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.04.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202104 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.10
Board used	Renesas Starter Kit+ for RX660 (product No.: RTK556609HCxxxxxBJ)

Table 6.5 Confirmed Operation Environment (Rev.2.20)

Item	Contents
Integrated development environment	Renesas Electronics e[2] studio Version 22.7.0 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.04.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202202 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.20
Board used	Renesas Starter Kit+ for RX671 (product No.: RTK55671EDCxxxxxBJ)

Table 6.6 Confirmed Operation Environment (Rev.2.30)

Item	Contents
Integrated development environment	Renesas Electronics e[2] studio Version 2022-10 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.05.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202204 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.30
Board used	Renesas Flexible Motor Control Kit for RX26T (Part Number: RTK0EMXE70S00020BJ)

Table 6.7 Confirmed Operation Environment (Rev.2.40)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 2023-04 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.05.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202204 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if "Optimize size (-Os)" is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.40
Board used	-

Table 6.8 Confirmed Operation Environment (Rev.2.50)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 2024-01.1 IAR Embedded Workbench for Renesas RX 5.10.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.06.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202311 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if "Optimize size (-Os)" is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 5.10.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.50
Board used	Renesas Flexible Motor Control Kit for RX26T (Part Number: RTK0EMXE70S00020BJ)

Table 6.9 Confirmed Operation Environment (Rev.2.60)

Item	Contents
Integrated development environment	Renesas Electronics e[2] studio Version 2024-07 IAR Embedded Workbench for Renesas RX 5.10.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.06.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202405 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 5.10.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.60
Board used	Evaluation Kit for RX261 (product No.: RTK5EK2610S00011BJ)

Table 6.10 Confirmed Operation Environment (Rev.2.70)

Item	Contents
Integrated development environment	Renesas Electronics e[2] studio Version 2024-10 IAR Embedded Workbench for Renesas RX 5.10.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.06.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202411 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 5.10.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.70
Board used	Renesas Starter Kit+ for RX671 (product No.: RTK55671xxxxxxxxxx)

Table 6.11 Confirmed Operation Environment (Rev.2.71)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 2025-01 IAR Embedded Workbench for Renesas RX 5.10.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.07.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202411 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 5.10.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.2.71
Board used	-

6.2 Troubleshooting

(1) Q: I have added the FIT module to the project and built it. Then I got the error: Could not open source file "platform.h".

A: The FIT module may not be added to the project properly. Check if the method for adding FIT modules is correct with the following documents:

- Using CS+:

  Application note "Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)"

- Using e^2 studio:

  Application note "Adding Firmware Integration Technology Modules to Projects (R01AN1723)"

When using this FIT module, the board support package FIT module (BSP module) must also be added to the project. Refer to the application note "Board Support Package Module Using Firmware Integration Technology (R01AN1685)".

(2) Q: I have added the FIT module to the project and built it. Then I got the error: This MCU is not supported by the current r_rsci_rx module.

A: The FIT module you added may not support the target device chosen in your project. Check the supported devices of added FIT modules.

(3) Q: I have added the FIT module to the project and built it. Then I got an error: ERROR - Unsupported channel chosen in r_rsci_config.h.

A: The setting in the file "r_rsci_rx_config.h" may be wrong. Check the file "r_rsci_rx_config.h". If there is a wrong setting, set the correct value for that. Refer to 2.8, Configuration Overview for details.

(4) Q: Transmit data is not output from the TXD pin.

A: The pin setting may not be performed correctly. When using this FIT module, the pin setting must be performed. Refer to 4. "Pin Setting" for details.

7. Reference Documents

User's Manual: Hardware
The latest versions can be downloaded from the Renesas Electronics website.

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

User's Manual: Development Tools
RX Family C/C++ Compiler CC-RX User's Manual (R20UT3248)

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

Related Technical Updates

This module reflects the content of the following technical updates.

TN-RX*-A0275A/E

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Mar.31.21	—	First release.
1.10	Sep.13.21	39 40 Program	Updated and added new demo project Added RSKRX671 to “5. Demo Projects” 6.1 Confirmed Operation Environment: Added Table for Rev.1.10. Fixed setting of transfer data direction in async mode.
2.00	Dec.03.21	1-40 25, 27, 37 46 Program	Added support for Manchester mode. Added support for transfer data direction and data invert in Async mode. 6.1 Confirmed Operation Environment: Added Table for Rev.2.00. Added support for transfer data direction and data invert in Async mode. Added support for Manchester mode.
2.10	Mar.31.22	1, 8 13-15 47 Program	Added support for RX660. Added code size corresponding to RX660. 6.1 Confirmed Operation Environment: Added Table for Rev.2.10. Added support for RX660.
2.20	Jul.29.22	48 Program	6.1 Confirmed Operation Environment: Added Table for Rev.2.20. Updated demo projects
2.30	Aug.15.22	1, 8, 10, 11 13, 15, 17 50 Program	Added support for RX26T. Added code size corresponding to RX26T. 6.1 Confirmed Operation Environment: Added Table for Rev.2.30. Added support for RX26T.
2.40	Jun.30.23	1 25, 46 51 Program	Added support for RX26T-256KB Deleted the description of FIT configurator from "2.13 Adding the FIT Module to Your Project", "4. Pin Settings" 6.1 Confirmed Operation Environment: Added Table for Rev.2.40. Added support for RX26T-256KB
2.50	Mar.29.24	51 53 Program	6.1 Confirmed Operation Environment: Added Table for Rev.2.50. Added TN-RX*-A0275A/E to “Related Technical Update”. Fixed RSCI (CH8, CH9) hardware bug for RX26T as per Renesas Technical Update TN-RX*-A0275A/E Rev.1.00.
2.60	Jun.28.24	1, 8, 10, 11 14, 16, 18 53 Program	Added support for RX260, RX261. Added code size corresponding to RX260, RX261. 6.1 Confirmed Operation Environment: Added Table for Rev.2.60. Added support for RX260, RX261. Added the source code to check unsupported channel for the supported device. Updated support command RSCI_CMD_SET_TXI_PRIORITY and RSCI_CMD_SET_RXI_PRIORITY in R_RSCI_Control() for RX260, RX261.

Rev.	Date	Description	
		Page	Summary
2.70	Dec.31.24	5	Added a description of nested interrupt.
		11	Added new macros RSCI_CFG_CHn_EN_TXI_NESTED_INT, RSCI_CFG_CHn_EN_RXI_NESTED_INT, RSCI_CFG_CHn_EN_TEI_NESTED_INT and RSCI_CFG_CHn_EN_ERI_NESTED_INT to support nested Interrupt.
		47	4. Pin Setting: Modified the following setting procedure: Please perform the pin setting after calling the R_RSCI_Open function.
		53	6.1 Confirmed Operation Environment: Added Table for Rev.2.70.
		Program	Added support nested interrupt.
2.71	Mar.15.25	54	6.1 Confirmed Operation Environment: Added Table for Rev.2.71.
		Program	Updated FIT Disclaimer and Copyright.

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

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

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.

2. Processing at power-on

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

3. Input of signal during power-off state

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

4. Handling of unused pins

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

5. Clock signals

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

6. Voltage application waveform at input pin

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

7. Prohibition of access to reserved addresses

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

8. Differences between products

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

Corporate Headquarters

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

Contact information

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

Trademarks