RX Family

SCI Multi-Mode Module Using Firmware Integration Technology

R01AN1815EJ0180
Rev. 1.80
Oct. 1, 2016

Introduction

This module provides Asynchronous, Master Synchronous, and Single Master Simple SPI (SSPI) support for all channels of the Serial Communications Interface (SCI) peripheral for supported RX Family MCUs. Channels and modes may be configured on an individual basis, with disabled channels and modes allocating no resources.

This module is hereinafter referred to as the “SCI FIT module”.

Target Device

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

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

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

Related Documents

- Firmware Integration Technology User’s Manual (R01AN1833)
- Board Support Package Firmware Integration Technology Module (R01AN1685)
- BYTEQ Firmware Integration Technology Module (R01AN1683)
- Adding Firmware Integration Technology Modules to Projects (R01AN1723)
- Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)

Contents

1. Overview

The SCI FIT module supports the following SCI peripheral functions depending on the RX MCU Groups.

Table 1.1 SCI Peripheral Functions Supported by MCU Groups

	SCIc	SCId	SCIe	SCIf	SCIg	SCIh	SCIi
RX110			✓	✓			
RX111			✓	✓			
RX113			✓	✓			
RX130					✓	✓	
RX210	✓	✓					
RX230	✓	✓					
RX231	✓	✓					
RX23T					✓		
RX24T					✓		
RX63N	✓	✓					
RX631	✓	✓					
RX64M					✓	✓	
RX65N					✓	✓	✓
RX71M					✓	✓	

It is recommended that you review the Serial Communications Interface chapter in the Hardware User's Manual for your specific RX Family MCU for full details on this peripheral circuit. All basic UART, Master SPI, and Master Synchronous mode functionality is supported by this driver. 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

Features not supported by this driver are:

- extended mode (channel 12)
- multiprocessor mode (all channels)
- event linking
- DMAC/DTC data transfer

Handling of Channels

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_sci_rx_config.h".

An individual channel is initialized in the application by calling R_SCI_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.

Interrupts, and Transmission and Reception

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 only used in Asynchronous mode. The TXI interrupt occurs when a byte 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_SCI_Open() call, it is called here with a TEI event passed to it. If it is to be included, the TEI interrupt must also be enabled for a channel though an R_SCI_Control() command. Support for TEI interrupts may be removed from the driver via a setting in "r_sci_rx_config.h".

The RXI interrupt occurs each time the RDR register has shifted in a byte. In Asynchronous mode, this byte is loaded into the receive circular buffer during the interrupt for access later via an R_SCI_Receive() call at the application level. If a callback function is provided, it is called with a receive event and the received byte. If the receive queue is full, it is called with a queue full event and the received/unstored byte. In SSPI and Synchronous modes, the shifted-in byte is loaded directly into the buffer specified from the last Receive() or SendReceive() call. The data received before Receive() or 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_SCI_Open function. Refer to 2.10 Parameters for details.

Error Detection

The ERI interrupt occurs when a framing, overrun, or parity error is detected by the receive device. 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. Whether a callback function is provided or not, the interrupt repeatedly writes 0 to the SSR error flag until the error condition is cleared.

1.1 SCI FIT Module

This module is implemented into the project as APIs. To implement this module, refer to 2.13 Adding Driver to Your Project.

1.2 Outline of APIs

Table 1.2 lists API functions included in this module. Section 2.9 Code Size lists memory sizes required to use this module.

Table 1.2 API Function List

Function	Description
R_SCI_Open()	Applies power to the SCI channel, initializes the associated registers, enables interrupts, and provides the channel handle for use with other API functions. Takes a callback function pointer for notifying the user at interrupt level whenever a receiver error or other interrupt events have occurred.
R_SCI_Close()	Removes power to the SCI channel and disables the associated interrupts.
R_SCI_Send()	Queues data and initiates transmit if transmitter is not in use.
R_SCI_Receive()	Fetches data from a queue which is filled by RXI interrupts.
R_SCI_SendReceive()	For Synchronous and SSPI modes only. Transmits and receives data simultaneously if the transceiver is not in use.
R_SCI_Control()	Handles special hardware or software operations for the SCI channel.
R_SCI_GetVersion()	Returns at runtime the driver version number.

2. API Information

This Driver API follows the Renesas API naming standards.

2.1 Hardware Requirements

This driver requires your MCU support the following features:

- SCI peripheral

2.2 Hardware Resource Requirements

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

2.2.1 SCI

This driver makes use of the SCI peripheral. Individual channels may be omitted by this driver by disabling them in the "r_sci_rx_config.h" file.

2.2.2 GPIO

This driver utilizes port pins corresponding to each individual channel. These pins may not be used for GPIO. For SSPI mode, additional port pins are needed to serve as Slave Select lines.

2.3 Software Requirements

This driver is dependent upon the following Firmware Integration Technology (FIT) modules:

- r_bsp
- r_byteq (Asynchronous mode only)

2.4 Limitations

The following limitation applies to this module.

- When using RX63N and RX631, the RSPI cannot be used simultaneously. This is because the vector setting for the SCI group interrupts is done in this driver and group interrupts used in SCI and RSPI can cause interrupt races.

2.5 Supported Toolchains

This driver is tested and working with the following toolchains:

- Renesas RX Toolchain v.2.04.01 (RX110, RX111, RX113, RX130, RX210, RX230, RX231, RX23T, RX24T, RX63N, RX631, RX64M, RX71M)
- Renesas RX Toolchain v.2.05.00 (RX65N)

2.6 Header Files

All API calls and their supporting interface definitions are located in r_sci_rx_if.h. Compile time configurable options are located in r_sci_rx_config.h. Both of these files should be included by the User's application.

2.7 Integer Types

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

2.8 Configuration Overview

All configurable options that can be set at build time are located in the file “r_sci_rx_config.h”. A summary of these settings are provided in the following table:

Table 2.1 Description of configuration options

Configuration options in *r_sci_rx_config.h (1/2)*	
`#define SCI_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.

`#define SCI_CFG_ASYNC_INCLUDED 1`
`#define SCI_CFG_SYNC_INCLUDED 0`
`#define SCI_CFG_SSPI_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.

`#define SCI_CFG_DUMMY_TX_BYTE 0xFF`	This #define is used only with SSPI and Synchronous mode. It is the value which is clocked out for each byte clocked in during a Receive() function call.

`#define SCI_CFG_CH0_INCLUDED 0`
`#define SCI_CFG_CH1_INCLUDED 1`
`#define SCI_CFG_CH2_INCLUDED 0`
`#define SCI_CFG_CH3_INCLUDED 0`
`#define SCI_CFG_CH4_INCLUDED 0`
`#define SCI_CFG_CH5_INCLUDED 0`
`#define SCI_CFG_CH6_INCLUDED 0`
`#define SCI_CFG_CH7_INCLUDED 0`
`#define SCI_CFG_CH8_INCLUDED 0`
`#define SCI_CFG_CH9_INCLUDED 0`
`#define SCI_CFG_CH10_INCLUDED 0`
`#define SCI_CFG_CH11_INCLUDED 0`
`#define SCI_CFG_CH12_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.
	Note that only CH1 is enabled by default. Be sure to enable the channels you will be using in the config file.

`#define SCI_CFG_CH0_TX_BUFSIZ 80`
`#define SCI_CFG_CH1_TX_BUFSIZ 80`
`#define SCI_CFG_CH2_TX_BUFSIZ 80`
`#define SCI_CFG_CH3_TX_BUFSIZ 80`
`#define SCI_CFG_CH4_TX_BUFSIZ 80`
`#define SCI_CFG_CH5_TX_BUFSIZ 80`
`#define SCI_CFG_CH6_TX_BUFSIZ 80`
`#define SCI_CFG_CH7_TX_BUFSIZ 80`
`#define SCI_CFG_CH8_TX_BUFSIZ 80`
`#define SCI_CFG_CH9_TX_BUFSIZ 80`
`#define SCI_CFG_CH10_TX_BUFSIZ 80`
`#define SCI_CFG_CH11_TX_BUFSIZ 80`
`#define SCI_CFG_CH12_TX_BUFSIZ 80`
	These #defines specify the size of the buffer to be used in Asynchronous mode for the transmit queue on each channel. If the corresponding SCI_CFG_CHn_INCLUDED is set to 0, or SCI_CFG_ASYNC_INCLUDED is set to 0, the buffer is not allocated.

Configuration options in *r_sci_rx_config.h (2/2)*	

`#define SCI_CFG_CH0_RX_BUFSIZ 80`
`#define SCI_CFG_CH1_RX_BUFSIZ 80`
`#define SCI_CFG_CH2_RX_BUFSIZ 80`
`#define SCI_CFG_CH3_RX_BUFSIZ 80`
`#define SCI_CFG_CH4_RX_BUFSIZ 80`
`#define SCI_CFG_CH5_RX_BUFSIZ 80`
`#define SCI_CFG_CH6_RX_BUFSIZ 80`
`#define SCI_CFG_CH7_RX_BUFSIZ 80`
`#define SCI_CFG_CH8_RX_BUFSIZ 80`
`#define SCI_CFG_CH9_RX_BUFSIZ 80`
`#define SCI_CFG_CH10_RX_BUFSIZ 80`
`#define SCI_CFG_CH11_RX_BUFSIZ 80`
`#define SCI_CFG_CH12_RX_BUFSIZ 80`
	These #defines specify the size of the buffer to be used in Asynchronous mode for the receive queue on each channel. If the corresponding SCI_CFG_CHn_INCLUDED is set to 0, or SCI_CFG_ASYNC_INCLUDED is set to 0, the buffer is not allocated.

`#define SCI_CFG_TEI_INCLUDED 0`	Setting this #define to 1 causes the Transmit Buffer Empty interrupt code to be included. An R_SCI_Control() command is used to enable TEI interrupts for a particular channel. This interrupt only occurs when the last bit of the last byte of data has been sent and the transmitter has become idle. The interrupt calls the user's callback function (specified in R_SCI_Open()) and passes it an SCI_EVT_TEI event.
`#define SCI_CFG_RXERR_PRIORITY 3`	RX63N/631 ONLY. This sets the Group12 receiver error interrupt priority level. 1 is the lowest priority and 15 is the highest. This interrupt handles overrun, framing, and parity errors for all channels.
`#define SCI_CFG_ERI_TEI_PRIORITY 3`	RX64M/71M ONLY. 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. The TEI interrupt indicates when the last bit has been transmitted and the transmitter is idle (Asynchronous mode).
`#define SCI_CFG_CH10_FIFO_INCLUDED 0` `#define SCI_CFG_CH11_FIFO_INCLUDED 0`	ONLY MCUs which has the SCI module (SCIi) with FIFO function. • 1: Processing regarding the FIFO function is included in the build • 0: processing regarding the FIFO function is omitted from the build
`#define SCI_CFG_CH10_TX_FIFO_THRESH 8` `#define SCI_CFG_CH11_TX_FIFO_THRESH 8`	ONLY MCUs which has the SCI module (SCIi) with FIFO function. When the SCI operating mode is clock synchronous mode or simple SPI mode, set the values same as the receive FIFO threshold value. • 0 to 15: Specifies the threshold value of the transmit FIFO.
`#define SCI_CFG_CH10_RX_FIFO_THRESH 8` `#define SCI_CFG_CH11_RX_FIFO_THRESH 8`	ONLY MCUs which has the SCI module (SCIi) with FIFO function. • 1 to 15: Specifies the threshold value of the receive FIFO.

2.9 Code Size

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

Since this software has dependencies on other FIT modules, the memory size of those FIT modules must be allowed for when using this software. To get the memory requirements for dependent FIT modules see the respective user documents for the modules listed in "2.3 Software Requirements"

The sizes shown in the following tables are for RX130, RX231, RX64M, and RX65N in each communication method.

ROM and RAM minimum sizes (bytes) (1/2)

Device	Communication method		Memory usage		Remarks
			With Parameter Checking	Without Parameter Checking	
RX130	Asynchronous mode	ROM	2759 bytes	2494 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	2529 bytes	2198 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	3788 bytes	3346 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		168 bytes		
RX231	Asynchronous mode	ROM	2692 bytes	2426 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	2415 bytes	2131 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	3741 bytes	3346 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		136 bytes		
RX64M	Asynchronous mode	ROM	2833 bytes	2863 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	2488 bytes	2204 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	3876 bytes	3467 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		136 bytes		

ROM and RAM minimum sizes (bytes) (2/2)					
Device	Communication method		Memory usage		Remarks
			With Parameter Checking	Without Parameter Checking	
RX65N	Asynchronous mode	ROM	2767 bytes	2521 bytes	1 channel used
		RAM	192 bytes	192 bytes	1 channel used
	Clock synchronous mode	ROM	2476 bytes	2192 bytes	1 channel used
		RAM	36 bytes	36 bytes	1 channel used
	Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	3875 bytes	3455 bytes	Total 2 channels used
		RAM	392 bytes	392 bytes	Total 2 channels used
	Maximum stack usage		136 bytes		
	FIFO mode + Asynchronous mode	ROM	3428 bytes	3084 bytes	1 channel used
		RAM	196 bytes	196 bytes	1 channel used
	FIFO mode + Clock synchronous mode	ROM	3473 bytes	2965 bytes	1 channel used
		RAM	40 bytes	40 bytes	1 channel used
	FIFO mode + Asynchronous mode + Clock synchronous mode (or simple SPI)	ROM	4999 bytes	4517 bytes	Total 2 channels used
		RAM	400 bytes	400 bytes	Total 2 channels used
	Maximum stack usage		152 bytes		

RAM requirements vary based on the number of channels configured. Each channel has associated data structures in RAM. In addition, for Asynchronous mode, each Async 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 mode RAM requirements is:

Number of channels used (1 to 12) × (Data structure per channel (32 bytes)
+ Transmit queue buffer size (size specified by SCI_CFG_CHn_TX_BUFSIZ)
+ Receive queue buffer size (size specified by SCI_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 slightly depending on the combination of channels selected and the effects of compiler code optimization.

2.10 Parameters

This section describes the structure which is the parameters of API functions. This structure is found in r_sci_rx_if.h along with the API prototype declarations.

Structure for Managing Channels

This structure is to store management information required to control SCI 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 RX65N:

```
typedef struct st_sci_ch_ctrl     // Channel management structure
{
       sci_ch_rom_t const *rom;   // Start address of the SCI register for the channel
       sci_mode_t mode;           // SCI 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 (SCI_CFG_ASYNC_INCLUDED)
       byteq_hdl_t que;           // Transmit byte queue (asynchronous mode)
#endif
       uint8_t *buf;              // Start address of the transmit buffer
                                  //(clock synchronous/SSPI mode)
       } u_tx_data;
       union
       {
#if (SCI_CFG_ASYNC_INCLUDED)
       byteq_hdl_t que;           // Receive byte queue (asynchronous 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 (SCI_CFG_SSPI_INCLUDED || SCI_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 SCI_CFG_FIFO_INCLUDED
       uint8_t fifo_ctrl;         // FIFO function (enable/disable)
       uint8_t rx_dflt_thresh;    // Receive FIFO threshold value
       uint8_t tx_dflt_thresh;    // Transmit FIFO threshold value
#endif
} sci_ch_ctrl_t;
```

2.11 Return Values

This shows the different values API functions can return. This enum is found in r_sci_rx_if.h along with the API function declarations.

```
typedef enum e_sci_err      // SCI API error codes
{
    SCI_SUCCESS=0,
    SCI_ERR_BAD_CHAN,       // Non-existent channel number
    SCI_ERR_OMITTED_CHAN,   // SCI_CHx_INCLUDED is 0 in config.h
    SCI_ERR_CH_NOT_CLOSED,  // Channel still running in another mode
    SCI_ERR_BAD_MODE,       // Unsupported or incorrect mode for channel
    SCI_ERR_INVALID_ARG,    // Argument is not valid for parameter
    SCI_ERR_NULL_PTR,       // Received null ptr; missing required argument
    SCI_ERR_XCVR_BUSY,      // Cannot start data transfer; transceiver busy

    // Asynchronous mode only
    SCI_ERR_QUEUE_UNAVAILABLE,  // Cannot open tx or rx queue or both
    SCI_ERR_INSUFFICIENT_SPACE, // Not enough space in transmit queue
    SCI_ERR_INSUFFICIENT_DATA,  // Not enough data in receive queue

    // Synchronous/SSPI modes only
    SCI_ERR_XFER_NOT_DONE   // Data transfer still in progress
} sci_err_t;
```

2.12 Callback Function

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 mode, and when a transmit end interrupt occurs.

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

```
typedef struct st_sci_cb_args     // Arguments of the callback function
{
       sci_hdl_t hdl;             // Handle upon an event occurrence
       sci_cb_evt_t event;        // Event which triggered the event occurred
       uint8_t byte;              // Receive data upon an event occurrence
} sci_cb_args_t;

typedef enum e_sci_cb_evt         // Event for the callback function
{
       // Events for asynchronous mode
       SCI_EVT_TEI,               // TEI interrupt occurred; Transmit device is idle.
       SCI_EVT_RX_CHAR,           // Character received; Have placed in the queue.
       SCI_EVT_RXBUF_OVFL,        // Receive queue full; No more data can be stored.
       SCI_EVT_FRAMING_ERR,       // Framing error occurred in the receiver.
       SCI_EVT_PARITY_ERR,        // Parity error occurred in the receiver.
       // Events for SSPI/clock synchronous mode
       SCI_EVT_XFER_DONE,         // Transfer completed.
       SCI_EVT_XFER_ABORTED,      // Transfer canceled.
       // Common event
       SCI_EVT_OVFL_ERR           // Overrun error; Hardware for the receive device
} sci_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.

When the SCI_EVT_TEI, SCI_EVT_XFER_DONE or SCI_ECT_XFER_ABORTED event occurs, or the FIFO function is enabled, a received data is not stored. The following shows an example template for the callback function in asynchronous mode.

```
void MyCallback(void *p_args)
{
       sci_cb_args_t *args;
       args = (sci_cb_args_t *)p_args;
       if (args->event == SCI_EVT_RX_CHAR)
       {
              //from RXI interrupt; character placed in queue is in args->byte
              nop();
       }
#if SCI_CFG_TEI_INCLUDED
       else if (args->event == SCI_EVT_TEI)
       {
              // from TEI interrupt; transmitter is idle
              // possibly disable external transceiver here
              nop();
       }
#endif
       else if (args->event == SCI_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 == SCI_EVT_OVFL_ERR)
       {
              // from ERI/Group12 interrupt; receiver overflow error occurred
              // error char is in args->byte
              // error condition is cleared in ERI routine
              nop();
       }
       else if (args->event == SCI_EVT_FRAMING_ERR)
       {
              // from ERI/Group12 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 == SCI_EVT_PARITY_ERR)
       {
              // from ERI/Group12 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)
{
       sci_cb_args_t *args;
       args = (sci_cb_args_t *)p_args;
       if (args->event == SCI_EVT_XFER_DONE)
       {
              // data transfer completed
              nop();
       }
       else if (args->event == SCI_EVT_XFER_ABORTED)
       {
              // data transfer aborted
              nop();
       }
       else if (args->event == SCI_EVT_OVFL_ERR)
       {
              // from ERI or Group12 (RX63x) interrupt; receiver overflow error
              occurred
              // error char is in args->byte
              // error condition is cleared in ERI/Group12 interrupt routine
              nop();
       }
}
```

2.13 Adding Driver to Your Project

The FIT module must be added to each project in the e^2 studio.

You can use the FIT plug-in to add the FIT module to your project, or the module can be added manually.

It is recommended to use the FIT plug-in as you can add the module to your project easily and also it will automatically update the include file paths for you.

To add the FIT module using the plug-in, refer to chapter 2. “Adding FIT Modules to e^2 studio Projects Using FIT Plug-In” in the “Adding Firmware Integration Technology Modules to Projects” application note (R01AN1723).

To add the FIT module manually, refer to chapter 3. “Adding FIT Modules to e^2 studio Projects Manually” in the “Adding Firmware Integration Technology Modules to Projects (R01AN1723)”

When using the FIT module, the BSP FIT module also needs to be added. For details on adding the BSP FIT module, refer to the “Board Support Package Module Using Firmware Integration Technology” application note (R01AN1685).

3. API Functions

3.1 R_SCI_Open()

This function applies power to the SCI 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

```
sci_err_t  R_SCI_Open(uint8_t  const        chan,
                      sci_mode_t  const     mode,
                      sci_cfg_t * const     p_cfg,
                      void                  (* const p_callback)(void *p_args),
                      sci_hdl_t * const     p_hdl);
```

Parameters

chan
Channel to initialize
mode
Operational mode (see enumeration below)
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.
p_hdl
Pointer to a handle for channel (value set here)
Confirm the return value from R_SCI_Open is “SCI_SUCCESS” and then set the first parameter for the other APIs except R_SCI_GetVersion(). Refer to 2.10 Parameters.

The following SCI 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_sci_mode     // SCI operational modes
{
    SCI_MODE_OFF=0,         // channel not in use
    SCI_MODE_ASYNC,         // Asynchronous
    SCI_MODE_SSPI,          // Simple SPI
    SCI_MODE_SYNC,          // Synchronous
    SCI_MODE_MAX            // End of modes currently supported
} sci_mode_t;
```

The following #defines indicate configurable options for Asynchronous mode used in its configuration structure. These values correspond to bit definitions in the SMR register. It should be noted that when bit 1 is set (use external clock), bit 0 becomes “don’t care” in the register. The driver makes use of this condition by using bit 0 to distinguish between an 8x and 16x external clock. When an 8x clock is in use, the ABCS bit in the SEMR register must be set by the driver. In addition, when using the channel 10 and 11 in the Synchronous mode or SSPI mode as 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.)

```
#define SCI_CLK_INT         0x00  // use internal clock for baud rate generation
#define SCI_CLK_EXT_8X      0x03  // use external clock 8x baud rate
#define SCI_CLK_EXT_16X     0x02  // use external clock 16x baud rate
#define SCI_DATA_7BIT       0x40
#define SCI_DATA_8BIT       0x00
#define SCI_PARITY_ON       0x20
#define SCI_PARITY_OFF      0x00
#define SCI_ODD_PARITY      0x10
#define SCI_EVEN_PARITY     0x00
#define SCI_STOPBITS_2      0x08
#define SCI_STOPBITS_1      0x00
```

The complete runtime configurable options for Asynchronous mode are declared in the structure below. This structure is an element of the p_cfg parameter.

```
typedef struct st_sci_uart
{
    uint32_t    baud_rate;      // ie 9600, 19200, 115200 (valid for internal clock)
    uint8_t     clk_src;        // use SCI_CLK_INT/EXT8/EXT16
    uint8_t     data_size;      // use SCI_DATA_nBIT
    uint8_t     parity_en;      // use SCI_PARITY_ON/OFF
    uint8_t     parity_type;    // use SCI_ODD/EVEN_PARITY
    uint8_t     stop_bits;      // use SCI_STOPBITS_1/2
    uint8_t     int_priority;   // txi, tei, rxi, eri INT priority; 1=low, 15=high
} sci_uart_t;
```

The following enumeration is used in the configuration structure when SSPI or Synchronous mode is specified.

```
typedef enum e_sci_spi_mode
{
    SCI_SPI_MODE_OFF = 1, // channel is in synchronous mode

    SCI_SPI_MODE_0 = 0x80,// SPMR Register CKPH=1, CKPOL=0
                          // Mode 0: 00 CPOL=0 resting lo, CPHA=0 leading edge/rising
    SCI_SPI_MODE_1 = 0x40,// SPMR Register CKPH=0, CKPOL=1
                          // Mode 1: 01 CPOL=0 resting lo, CPHA=1 trailing edge/falling
    SCI_SPI_MODE_2 = 0xC0,// SPMR Register CKPH=1, CKPOL=1
                          // Mode 2: 10 CPOL=1 resting hi, CPHA=0 leading edge/falling
    SCI_SPI_MODE_3 = 0x00 // SPMR Register CKPH=0, CKPOL=0
                          // Mode 3: 11 CPOL=1 resting hi, CPHA=1 trailing edge/rising
} sci_spi_mode_t;
```

The configuration structure for SSPI and Synchronous modes is as follows:

```
typedef struct st_sci_sync_sspi
{
    sci_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
} sci_sync_sspi_t;
```

The union for p_cfg is:

```
typedef union
{
    sci_uart_t      async;
    sci_sync_sspi_t sync;
    sci_sync_sspi_t sspi;
} sci_cfg_t;
```

Return Values

SCI_SUCCESS:	*Successful; channel initialized*
SCI_ERR_BAD_CHAN:	*Channel number is invalid for part*
SCI_ERR_OMITTED_CHAN:	*Corresponding SCI_CHx_INCLUDED is 0*
SCI_ERR_CH_NOT_CLOSED:	*Channel currently in operation; Perform R_SCI_Close() first*
SCI_ERR_BAD_MODE:	*Specified mode not currently supported*
SCI_ERR_NULL_PTR:	*p_cfg pointer is NULL*
SCI_ERR_INVALID_ARG:	*An element of the p_cfg structure contains an invalid value.*
SCI_ERR_QUEUE_UNAVAILABLE:	*Cannot open transmit or receive queue or both (Asynchronous mode)*

Properties

Prototyped in file “r_sci_rx_if.h”

Description

Initializes an SCI 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 mode. See Control() for enabling TEI interrupts.

Reentrant

Function is re-entrant for different channels.

Example: Asynchronous Mode

```
sci_cfg_t   config;
sci_hdl_t   Console;
sci_err_t   err;

config.async.baud_rate = 115200;
config.async.clk_src = SCI_CLK_INT;
config.async.data_size = SCI_DATA_8BIT;
config.async.parity_en = SCI_PARITY_OFF;
config.async.parity_type = SCI_EVEN_PARITY;   // ignored because parity is disabled
config.async.stop_bits = SCI_STOPBITS_1;
config.async.int_priority = 2;                // 1=lowest, 15=highest

err = R_SCI_Open(SCI_CH1, SCI_MODE_ASYNC, &config, MyCallback, &Console);
```

Example: SSPI Mode

```
sci_cfg_t   config;
sci_hdl_t   sspiHandle;
sci_err_t   err;

config.sspi.spi_mode = SCI_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_SCI_Open(SCI_CH12, SCI_MODE_SSPI, &config, sspiCallback, &sspiHandle);
```

Example: Synchronous Mode

```
sci_cfg_t   config;
sci_hdl_t   syncHandle;
sci_err_t   err;

config.sync.spi_mode = SCI_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_SCI_Open(SCI_CH12, SCI_MODE_SYNC, &config, syncCallback, &syncHandle);
```

Special Notes:

The driver uses an algorithm for calculating the optimum values for BRR, SEMR.ABCS, and SMR.CKS using 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_SCI_Open() function, and the pin function and mode must be selected after calling the R_SCI_Open() function. The following is an example initialization for RX111 channel 1:

Before the R_SCI_Open() function call

```
PORT1.PDR.BIT.B7 = 0;       // set SCK pin direction to input (dflt)
```

After the R_SCI_Open() function call

```
MPC.P17PFS.BYTE = 0x0A;     // Pin Func Select P17 SCK1
PORT1.PMR.BIT.B7 = 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_SCI_Open() function, and the pin functions and modes must be selected after calling the R_SCI_Open() function. An example for initializing channel 6 for SSPI on the RX64M is as follows:

Before the R_SCI_Open() function call

```
PORT0.PODR.BIT.B2 = 0;      // set line low
PORT0.PODR.BIT.B0 = 0;      // set line low
PORT0.PDR.BIT.B2 = 1;       // set clock pin direction to output
PORT0.PDR.BIT.B0 = 1;       // set MOSI pin direction to output
PORT0.PDR.BIT.B1 = 0;       // set MISO pin direction to input
```

After the R_SCI_Open() function call

```
MPC.P00PFS.BYTE = 0x0A;     // Pin Func Select P00 MOSI
MPC.P01PFS.BYTE = 0x0A;     // Pin Func Select P01 MISO
MPC.P02PFS.BYTE = 0x0A;     // Pin Func Select P02 SCK
PORT0.PMR.BIT.B0 = 1;       // set MOSI pin mode to peripheral
PORT0.PMR.BIT.B1 = 1;       // set MISO pin mode to peripheral
PORT0.PMR.BIT.B2 = 1;       // set clock pin mode to peripheral
```

3.2 R_SCI_Close()

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

Format
sci_err_t R_SCI_Close(sci_hdl_t const hdl);

Parameters
hdl
Handle for channel
Set *hdl* when R_SCI_Open() is successfully processed.

Return Values
SCI_SUCCESS: *Successful; channel closed*
SCI_ERR_NULL_PTR: *hdl is NULL*

Properties
Prototyped in file “r_sci_rx_if.h”

Description
Disables the SCI channel designated by the handle. Does not free any resources but saves power and allows the corresponding channel to be re-opened later, potentially with a different configuration.

Reentrant
Function is re-entrant for different channels.

Example
```
sci_hdl_t   Console;
  ...
err = R_SCI_Open(SCI_CH1, SCI_MODE_ASYNC, &config, MyCallback, &Console);
  ...
err = R_SCI_Close(Console);
```

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

3.3 R_SCI_Send()

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

Format

```
sci_err_t   R_SCI_Send(sci_hdl_t const    hdl,
                       uint8_t            *p_src,
                       uint16_t const     length);
```

Parameters

hdl

Handle for channel

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

p_src

Pointer to data to transmit

length

Number of bytes to send

Return Values

SCI_SUCCESS:	*Transmit initiated or loaded into queue (Asynchronous)*
SCI_ERR_NULL_PTR:	*hdl value is NULL*
SCI_ERR_BAD_MODE:	*Channel mode not currently supported*
SCI_ERR_INSUFFICIENT_SPACE:	*Insufficient space in queue to load all data (Asynchronous)*
SCI_ERR_XCVR_BUSY:	*Channel currently busy (SSPI/Synchronous)*

Properties

Prototyped in file “r_sci_rx_if.h”

Description

In Asynchronous mode, this function places data into a transmit queue for sending on an SCI channel referenced by the handle. Transmission begins immediately if another byte is not already being sent. Be noticed that when using a channel in FIFO mode, a R_SCI_Send() function should not be called when a transmission is in process. The completion of transmission process can be confirmed by the occurrence of transmit end interrupt. To enable transmit end interrupt, the following settings are required.

- Set SCI_CFG_TEI_INCLUDED in r_sci_rx_config.h to 1.
- Enable transmit end interrupt by R_SCI_Control() function.

In SSPI and Synchronous modes, no data is queued and transmission begins immediately if the transceiver is not already in use. All transmissions are handled at the interrupt level.

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.

Reentrant

Function is re-entrant for different channels.

Example: Asynchronous Mode

```
#define STR_CMD_PROMPT "Enter Command: "
sci_hdl_t Console;
sci_err_t err;

err = R_SCI_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

```
sci_hdl_t  sspiHandle;
sci_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_SCI_Send(sspiHandle, &flash_cmd, 1);
while (SCI_SUCCESS != R_SCI_Control(sspiHandle, SCI_CMD_CHECK_XFER_DONE, NULL))
{
}

/* READ ID FROM FLASH DEVICE */
R_SCI_Receive(sspiHandle, sspi_buf, 5);
while (SCI_SUCCESS != R_SCI_Control(sspiHandle, SCI_CMD_CHECK_XFER_DONE, NULL))
{
}

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

Example: Synchronous Mode

```
#define STRING1 "Test String"
sci_hdl_t  lcdHandle;
sci_err_t  err;


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

while (SCI_SUCCESS != R_SCI_Control(lcdHandle, SCI_CMD_CHECK_XFER_DONE, NULL))
{
}
```

Special Notes:

None.

3.4 R_SCI_Receive()

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

Format

```
sci_err_t  R_SCI_Receive(sci_hdl_t const   hdl,
                         uint8_t           *p_dst,
                         uint16_t const    length);
```

Parameters

hdl
Handle for channel
Set *hdl* when R_SCI_Open() is successfully processed.
p_dst
Pointer to buffer to load data into
length
Number of bytes to read

Return Values

SCI_SUCCESS:	*Requested number of bytes were loaded into p_dst (Asynchronous)* *Clocking in of data initiated (SSPI/Synchronous)*
SCI_ERR_NULL_PTR:	*hdl value is NULL*
SCI_ERR_BAD_MODE:	*Channel mode not currently supported*
SCI_ERR_INSUFFICIENT_DATA:	*Insufficient data in receive queue to fetch all data (Asynchronous)*
SCI_ERR_XCVR_BUSY:	*Channel currently busy (SSPI/Synchronous)*

Properties

Prototyped in file “r_sci_rx_if.h”

Description

In Asynchronous mode, this function gets data received on an SCI 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 SCI_CFG_DUMMY_TX_BYTE in r_sci_config.h is clocked out while the receive data is being clocked in.

If any errors occurred during reception by hardware, they are handled by the callback function specified in R_SCI_Open() and no corresponding error code is provided here.

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.

Reentrant

Function is re-entrant for different channels.

Example: Asynchronous Mode

```
sci_hdl_t Console;
sci_err_t err;
uint8_t   byte;

/* echo characters */
while (1)
{
    while (SCI_SUCCESS != R_SCI_Receive(Console, &byte, 1))
    {
    }
    R_SCI_Send(Console, &byte, 1);
}
```

Example: SSPI Mode

```
sci_hdl_t  sspiHandle;
sci_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_SCI_Send(sspiHandle, &flash_cmd, 1);
while (SCI_SUCCESS != R_SCI_Control(sspiHandle, SCI_CMD_CHECK_XFER_DONE, NULL))
{
}

/* READ ID FROM FLASH DEVICE */
R_SCI_Receive(sspiHandle, sspi_buf, 5);
while (SCI_SUCCESS != R_SCI_Control(sspiHandle, SCI_CMD_CHECK_XFER_DONE, NULL))
{
}

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

Example: Synchronous Mode

```
sci_hdl_t  sensorHandle;
sci_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_SCI_Send(sensorHandle, &sensor_cmd, 1);
while (SCI_SUCCESS != R_SCI_Control(sensorHandle, SCI_CMD_CHECK_XFER_DONE, NULL))
{
}

/* READ LEVEL FROM SENSOR */
R_SCI_Receive(sensorHandle, sync_buf, 4);
while (SCI_SUCCESS != R_SCI_Control(sensorHandle, SCI_CMD_CHECK_XFER_DONE, NULL))
{
}
```

Special Notes:

See section 2.12 Callback Function for error reporting with a callback function.

3.5 R_SCI_SendReceive()

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

Format

```
sci_err_t  R_SCI_SendReceive(sci_hdl_t const   hdl,
                             uint8_t           *p_src,
                             uint8_t           *p_dst,
                             uint16_t const    length);
```

Parameters

hdl
Handle for channel
Set *hdl* when R_SCI_Open() is successfully processed.

p_src
Pointer to data to transmit

p_dst
Pointer to buffer to load data into

length
Number of bytes to send

Return Values

SCI_SUCCESS:	*Data transfer initiated*
SCI_ERR_NULL_PTR:	*hdl value is NULL*
SCI_ERR_BAD_MODE:	*Channel mode not SSPI or Synchronous*
SCI_ERR_XCVR_BUSY:	*Channel currently busy*

Properties

Prototyped in file “r_sci_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.

Reentrant

Function is re-entrant for different channels.

Example: SSPI Mode

```
sci_hdl_t  sspiHandle;
sci_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, SCI_CFG_DUMMY_TX_BYTE };

FLASH_SS = SS_ON;

err = R_SCI_SendReceive(sspiHandle, out_buf, in_buf, 2);
while (SCI_SUCCESS != R_SCI_Control(sspiHandle, SCI_CMD_CHECK_XFER_DONE, NULL))
{
}

FLASH_SS = SS_OFF;

// in_buf[1] contains status
```

Special Notes:
See section 2.12 Callback Function for error reporting with a callback function.

3.6 R_SCI_Control()

This function handles special hardware and software operations for the SCI channel.

Format

```
sci_err_t  R_SCI_Control (sci_hdl_t const     hdl,
                          sci_cmd_t const     cmd,
                          void                *p_args);
```

Parameters

hdl

Handle for channel

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

cmd

Command to run (see enumeration below)

p_args

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

The valid *cmd* values are as follows:

```
typedef enum e_sci_cmd          // SCI Control() commands
{
    // All modes
    SCI_CMD_CHANGE_BAUD,        // change baud/bit rate
    SCI_CMD_CHANGE_TX_FIFO_THRESH,      // change transmit FIFO threshold value
    SCI_CMD_CHANGE_RX_FIFO_THRESH,      // change receive FIFO threshold value

    // Async commands
    SCI_CMD_EN_NOISE_CANCEL,    // enable noise cancellation
    SCI_CMD_EN_TEI,             // enable TEI interrupts
    SCI_CMD_OUTPUT_BAUD_CLK,    // output baud clock on the SCK pin
    SCI_CMD_START_BIT_EDGE,     // detect start bit as falling edge of RXDn pin
                                // (default detect as low level on RXDn pin)
    SCI_CMD_GENERATE_BREAK,     // generate break condition
    SCI_CMD_TX_Q_FLUSH,         // flush transmit queue
    SCI_CMD_RX_Q_FLUSH,         // flush receive queue
    SCI_CMD_TX_Q_BYTES_FREE,    // get count of unused transmit queue bytes
    SCI_CMD_RX_Q_BYTES_AVAIL_TO_READ, // get num bytes ready for reading

    // Async/Sync commands
    SCI_CMD_EN_CTS_IN,          // enable CTS input (default RTS output)

    // SSPI/Sync commands
    SCI_CMD_CHECK_XFER_DONE,  // see if send, rcv, or both are done; SCI_SUCCESS if yes
    SCI_CMD_ABORT_XFER,
    SCI_CMD_XFER_LSB_FIRST,
    SCI_CMD_XFER_MSB_FIRST,
    SCI_CMD_INVERT_DATA,

    // SSPI commands
    SCI_CMD_CHANGE_SPI_MODE
} sci_cmd_t;
```

Most of the commands do not require arguments and take NULL for p_args. The argument structure for SCI_CMD_CHANGE_BAUD is shown below.

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

The argument for SCI_CMD_TX_Q_BYTES_FREE and SCI_CMD_RX_Q_BYTES_AVAIL_TO_READ is a pointer to a uint16_t variable to hold a count value.

The argument for SCI_CMD_CHANGE_SPI_MODE is a pointer to an enumeration variable containing the new mode desired.

The argument for SCI_CMD_SET_TXI_PRIORITY and SCI_CMD_SET_RXI_PRIORITY (RX64M/RX71M) is a pointer to a uint8_t variable to hold the priority level. The transmit and receive interrupt priority levels are set to the same value by default in Open().

Return Values

SCI_SUCCESS:	*Successful; channel initialized*
SCI_ERR_NULL_PTR:	*hdl or p_args pointer is NULL (when required)*
SCI_ERR_BAD_MODE:	*Channel mode not currently supported*
SCI_ERR_INVALID_ARG:	*The cmd value or an element of p_args contains an invalid value.*

Properties

Prototyped in file “r_sci_rx_if.h”

Description

This function is used for configuring “non-standard” hardware features, changing driver configuration, and obtaining driver status.

By default, the SCI hardware outputs a low level on the RTSn#CTSn# pin when the receiver is available for receiving data. Here the pin functions as an RTS output signal. By issuing an SCI_CMD_EN_CTS_IN, the pin accepts CTS input signals. In this case, when the RTSn#CTSn# pin is high, the transmitter is disabled until the line transitions low again. If the transmitter is in process of sending a byte when the line goes high, it completes transmission of the byte before halting.

Reentrant

Function is re-entrant for different channels.

Example 1: Asynchronous Mode

```
sci_hdl_t   Console;
sci_cfg_t   config;
sci_baud_t  baud;
sci_err_t   err;
uint16_t    cnt;

R_SCI_Open(SCI_CH1, SCI_MODE_ASYNC, &config, MyCallback, &Console);
R_SCI_Control(Console, SCI_CMD_EN_NOISE_CANCEL, NULL);
R_SCI_Control(Console, SCI_CMD_EN_TEI, NULL);
  ...
/* reset baud rate due to low power mode clock switching */
baud.pclk = 8000000;      // 8 MHz
baud.rate = 19200;
R_SCI_Control(Console, SCI_CMD_CHANGE_BAUD, (void *)&baud);
  ...
/* after sending several messages, determine how much space is left in tx queue */
R_SCI_Control(Console, SCI_CMD_TX_Q_BYTES_FREE, (void *)&cnt);
  ...
/* check to see if there is data sitting in the receive queue */
R_SCI_Control(Console, SCI_CMD_RX_Q_BYTES_AVAIL_TO_READ, (void *)&cnt);
```

Example 2: SSPI Mode

```
sci_cfg_t   config;
sci_spi_mode_t   mode;
sci_hdl_t   sspiHandle;
sci_err_t   err;

config.sspi.spi_mode     = SCI_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_SCI_Open(SCI_CH12, SCI_MODE_SSPI, &config, sspiCallback, &sspiHandle);
  ...
  ...
// for changing to slave device which operates in a different mode
mode = SCI_SPI_MODE_3;
R_SCI_Control(sspiHandle, SCI_CMD_CHANGE_SPI_MODE, (void *)&mode);
```

Special Notes:

The driver uses an algorithm for calculating the optimum values for BRR, SEMR.ABCS, and SMR.CKS. This however does not guarantee a low bit error rate for all peripheral clock/baud rate combinations.

If the command SCI_CMD_EN_CTS_IN is to be used, the pin direction must be selected before calling the R_SCI_Open() function, and the pin function and mode must be selected after calling the R_SCI_Open() function. The following is an example initialization for RX111 channel 1:

Before the R_SCI_Open() function call

```
PORT1.PDR.BIT.B4 = 0;       // set CTS/RTS pin direction to input (dflt)
```

After the R_SCI_Open() function call

```
MPC.P14PFS.BYTE = 0x0B;     // Pin Func Select P14 CTS
PORT1.PMR.BIT.B4 = 1;       // set CTS/RTS pin mode to peripheral
```

If the command SCI_CMD_OUTPUT_BAUD_CLK is to be used, the pin direction must be selected before calling the R_SCI_Open() function, and the pin function and mode must be selected after calling the R_SCI_Open() function. The following is an example initialization for RX111 channel 1:

Before the R_SCI_Open() function call

```
PORT1.PDR.BIT.B7 = 1;       // set SCK pin direction to output
```

After the R_SCI_Open() function call

```
MPC.P17PFS.BYTE = 0x0A;     // Pin Func Select P17 SCK1
PORT1.PMR.BIT.B7 = 1;       // set SCK pin mode to peripheral
```

3.7 R_SCI_GetVersion()

This function returns the driver version number at runtime.

Format
uint32_t R_SCI_GetVersion(void)

Parameters
None

Return Values
Version number.

Properties
Prototyped in file “r_sci_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.

Reentrant
Yes

Example
```
uint32_t   version;
  ...
version = R_SCI_GetVersion();
```

Special Notes:
This function is inlined using the “#pragma inline” directive

4. Pin Setting

Regarding the pin setting used for this module, it is strictly recommended to perform the pin setting after calling the function R_SCI_Open.

5. Demo Projects

Demo projects are complete stand-alone programs. They include function main() that utilizes the module and its dependent modules (e.g. r_bsp). The standard naming convention for the demo project is <module>_demo_<board> where <module> is the peripheral acronym (e.g. s12ad, cmt, sci) and the <board> is the standard RSK (e.g. rskrx113). For example, s12ad FIT module demo project for RSKRX113 will be named as s12ad_demo_rskrx113. Similarly the exported .zip file will be <module>_demo_<board>.zip. For the same example, the zipped export/import file will be named as s12ad_demo_rskrx113.zip

5.1 sci_demo_rskrx113

This is a simple demo of the RX113 Serial Communications Interface (SCI) for the RSKRX113 starter kit (FIT module "r_sci_rx"). In the demo project, the MCU communicates with the terminal through the SCI channel configured as the UART. The RS232 interface is not on the RSKRX113 in the demo, thus the USB virtual COM interface is used as serial interface for RSKRX113. A PC running the terminal emulation application is required for communicating with the user.

Setup and Execution

1. Prepare jumpers for the RSKRX113 board. Mount J15 jumper between 1 and 2, and J16 jumper between 2 and 3.
2. Build this sample application, download it to the RSK board, and execute the application using a debugger.
3. Connect the serial port on the RSK board to the serial port on the PC.

   This demo project 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.
4. 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.
5. 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
6. 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.
7. This application is in echo mode. A given key input to the terminal is received by the SCI driver and then the application returns the characters to the terminal.

Boards Supported

RSKRX113

5.2 sci_demo_rskrx231

This is a simple demo of the RX231 Serial Communications Interface (SCI) for the RSKRX231 starter kit (FIT module "r_sci_rx"). In the demo project, the MCU communicates with the terminal through the SCI channel configured as the UART. The RS232 interface is not on the RSKRX231 in the demo, thus the USB virtual COM interface is used as serial interface for RSKRX231. 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 project 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 SCI driver and then the application returns the characters to the terminal.

Boards Supported

RSKRX231

5.3 sci_demo_rskrx64m

This is a simple demo of the RX64M Serial Communications Interface (SCI) for the RSKRX64M starter kit (FIT module "r_sci_rx"). In the demo project, the MCU communicates with the terminal through the SCI channel configured as the UART. The RS232 interface is not on the RSKRX64M in the demo, thus the USB virtual COM interface is used as serial interface for RSKRX64M. A PC running the terminal emulation application is required for communicating with the user.

Setup and Execution

1. Prepare jumpers for RSKRX64M board. Mount J16 and J18 jumpers between 2 and 3.
2. Build this sample application, download it to the RSK board, and execute the application using a debugger.
3. Connect the serial port on the RSK board to the serial port on the PC.

   This demo project 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.
4. 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.
5. 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
6. 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.
7. This application is in echo mode. A given key input to the terminal is received by the SCI driver and then the application returns the characters to the terminal.

Boards Supported

RSKRX64M

5.4 sci_demo_rskrx71m

This is a simple demo of the RX71M Serial Communications Interface (SCI) for the RSKRX71M starter kit (FIT module "r_sci_rx"). In the demo project, the MCU communicates with the terminal through the SCI channel configured as the UART. The RS232 interface is not on the RSKRX71M in the demo, thus the USB virtual COM interface is used as serial interface for RSKRX71M. A PC running the terminal emulation application is required for communicating with the user.

Setup and Execution

1. Prepare jumpers for RSKRX71M board. Mount J16 and J18 jumpers between 2 and 3.
2. Build this sample application, download it to the RSK board, and execute the application using a debugger.
3. 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.
4. 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.
5. 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
6. 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.
7. This application is in echo mode. A given key input to the terminal is received by the SCI driver and then the application returns the characters to the terminal.

Boards Supported

RSKRX71M

5.5 Adding a Demo to a Workspace

Demo projects are found in the FITDemos subdirectory of the distribution file for this application note. To add a demo project to a workspace, select File>Import>General>Existing Projects into Workspace, then click "Next". From the Import Projects dialog, choose the "Select archive file" radio button. "Browse" to the FITDemos subdirectory, select the desired demo zip file, then click "Finish".

6. Modules Provided

The modules provided can be downloaded from the Renesas Electronics website.

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*-A151A/E

Website and Support

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

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

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Nov-15-2013	—	Initial Multi-Mode Release.
1.20	Apr-17-2014	1,3	Added mention of RX110 support.
1.30	Jul-02-2014	-	Fixed RX63N bug that prevented receive errors (Group12) from interrupting except on channel 2.
1.40	Dec-16-2014	1, 7	Added RX113 to list of supported devices. Added section 2.11 Code Size and RAM usage.
1.50	Mar-18-2015	1,3,5	Added RX64M, RX71M to list of supported devices.
1.60	Jun-30-2015	1,3,5	Added RX231 to list of supported devices.
1.70	Sep-30-2015	— 7 11 13 22 22	Added support for the RX23T Group. Updated information of 2.11 Code Size and RAM usage including code sizes in Table 2. Modified the setting procedure in the following sections: • Special Notes in R_SCI_Open(): - When an external clock is used in Asynchronous mode - For settings of the pins used for communications • Special Notes in R_SCI_Control(): - When the command SCI_CMD_EN_CTS_IN is to be used - When the command SCI_CMD_OUTPUT_BAUD_CLK is to be used
1.80	Oct-1-2016	— 3, 4 5 7 8, 9 10 11 11-13 14 18, 19, 21 25 29 30-33 34	Added support for the RX65N Group. Revised the contents in 1. Overview including new sections 1.1 and 1.2. Added the limitation in 2.4 Limitations. Updated the information in 2.5 Supported Toolchains. Added the definitions regarding FIFO as the configuration option in Table 2.1. Updated the table for ROM and RAM minimum sizes and the formula in 2.9 Code Size. Added section 2.10 Parameters. Moved section Return Values from 3.2 to 2.11. Added section 2.12 Callback Function. 3.1 R_SCI_Open() - Added some descriptions in the introduction of the function, and parameters p_callback and p_hdl. - Moved descriptions regarding the callback function to section 2.12. 3.2 R_SCI_Close(),3.3 R_SCI_Send(),3.4 R_SCI_Receive(), 3.5 R_SCI_SendReceive() - Added a description in the parameter hdl. 3.6 R_SCI_Control() - Added a description in the parameter hdl. - Added definitions regarding FIFO in the valid cmd values. Added section 4. Pin Setting. Added section 5. Demo Projects. Added the section for technical update information.

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. Handling of Unused Pins

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

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

2. Processing at Power-on

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

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

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

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

4. Clock Signals

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

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

5. Differences between Products

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

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

Notice

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

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

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

RENESAS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Colophon 5.0