RX Family

SCI FIFO Module Using Firmware Integration Technology

Introduction

This application note describes the SCI FIFO module which uses Firmware Integration Technology (FIT).

This module provides Asynchronous and Master Synchronous support for all channels of the SCI FIFO peripheral. Channels and modes may be configured on an individual basis, with disabled channels and modes allocating no resources.

Target Device

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

- **RX64M 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)
- RX Family Board Support Package Firmware Integration Technology Module (R01AN1685)

Contents

1. Overview

1.1 SCIF FIT Module

The SCIF 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 SCIF FIT Module

This SCI FIFO driver supports the SCIFA peripheral on the RX64M and RX71M. The hardware functionality is detailed in Chapter 41 of the RX64M/RX71M Hardware User's Manual. All basic UART and Master Synchronous mode functionality is supported by this driver. Additionally, the driver supports the following features in Asynchronous mode:

- noise cancellation.
- MSB-first bit order
- flow control with CTS / RTS.

Features not supported by this driver are:

- DRIF interrupt (works only for messages less than threshold number of bytes in length)

This is a multi-channel driver which supports all channels present on the peripheral. Specific channels can be excluded via compile-time equates to reduce driver RAM and ROM usage and code size if desired. These equates are specified in "r_scif_rx_config.h".

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

This driver is interrupt-driven and non-blocking. For Asynchronous mode, data will be stored in the receive FIFO until an overflow occurs or an R_SCIF_Receive() is issued (whichever comes first). Interrupts supported by this driver are TXIF, RXIF, and the GROUPAL0 TEIF, ERIF, and BRIF interrupts.

The TXIF interrupt occurs whenever the configured threshold number of bytes remain in the transmit FIFO. During this interrupt the FIFO is loaded with more bytes from the transmit message until either no more data remains in the message or the transmit FIFO becomes full (whichever comes first). The TEIF interrupt occurs only after the last bit of the last byte from the FIFO has been shifted out of the TSR register. If a callback function is provided in the R_SCIF_Open() call, it is called here with a SCIF_EVT_TX_DONE (Asynchronous) or SCIF_EVT_XCV_DONE (Synchronous) event passed to it. The Send() and SendReceive() functions can have two transmit requests outstanding at a time to provide continuous streaming of data. The DONE event does not occur until all outstanding requests have been processed. If it is desired to know when each message completes, no more than one request should be outstanding at a time.

The RXIF interrupt occurs each time the receive FIFO contains the configured number of threshold bytes. During this interrupt, the message buffer is loaded with data from the FIFO until the requested number of bytes have been read or until no more data remains in the FIFO. When the entire number of bytes requested have been read and if a callback function is provided, it is called with a SCIF_EVT_RX_DONE (Asynchronous) or SCIF_EVT_XCV_DONE (Synchronous) event. The Receive() and SendReceive() functions can have two receive requests outstanding at a time to provide continuous streaming of data. The DONE event does not occur until all outstanding requests have been processed. If it is desired to know when each message completes, no more than one request should be outstanding at a time.

In Asynchronous mode, the ERIF interrupt occurs when a framing or parity error is detected by the receiver hardware, and the BRIF interrupt occurs when a Break is received or a receive-FIFO overflow occurs. If a callback function is provided, the interrupt determines which error occurred and notifies the application of the event. Whether a callback function is provided or not, the interrupt clears the error condition by writing “0” to the appropriate FSR or LSR error flag.

1.3 API Overview

Table 1.1 lists the API functions included in this module.

Table 1.1 API Functions

Function	Description
R_SCIF_Open	Applies power to the SCIF channel, initializes the associated registers, enables interrupts, and provides the channel handle for use with other API functions. Takes an optional callback function pointer for notifying the user at interrupt level whenever a receiver error or other interrupt events have occurred.
R_SCIF_Close	Removes power to the SCIF channel and disables the associated interrupts.
R_SCIF_Send	Queues message for sending on the transmit FIFO. Up to two requests can be outstanding at a time. Transmission begins immediately if transmitter is idle.
R_SCIF_Receive	Queues message for receiving from the receive FIFO. Up to two requests can be outstanding at a time. In Sync mode, driver starts clocking in data immediately if transceiver is idle.
R_SCIF_SendReceive	For Synchronous mode only. Transmits and receives data simultaneously. Up to two requests total (Send(), Receive(), and/or SendReceive()) can be outstanding at a time.
R_SCIF_Control	Handles special hardware or software operations for the SCIF channel.
R_SCIF_GetVersion	Returns at runtime the driver version number.

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:

- SCIFA 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 SCIFA

This driver makes use of the SCIFA peripheral. Individual channels may be omitted by this driver by disabling them in the “r_scif_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.

2.3 Software Requirements

This driver is dependent upon the following FIT module:

- Renesas Board Support Package (r_bsp)

2.4 Supported Toolchains

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

2.5 Interrupt Vector

For asynchronous mode, when the R_SCIF_Open function is executed, interrupts TXIFn, RXIFn, TEIFn, ERIFn, and BRIFn become enabled. When the R_SCIF_Send function is executed, the DRIFn interrupt becomes enabled.
For synchronous mode, when the R_SCIF_Open function is executed, interrupts TXIFn, RXIFn, and TEIFn become enabled. When the R_SCIF_Send function is executed, the DRIFn interrupt becomes enabled.
Table 2.1 shows the interrupt vectors used by the SCI FIFO FIT module.

Table 2.1 List of Usage of Interrupt Vectors

Device	Contents
RX64M RX71M	RXIF8 interrupt [channel 8] (vector no.: 100) TXIF8 interrupt [channel 8] (vector no.: 101) RXIF9 interrupt [channel 9] (vector no.: 102) TXIF9 interrupt [channel 9] (vector no.: 103) RXIF10 interrupt [channel 10] (vector no.: 104) TXIF10 interrupt [channel 10] (vector no.: 105) RXIF11 interrupt [channel 11] (vector no.: 114) TXIF11 interrupt [channel 11] (vector no.: 115) GROUPAL0 interrupt (vector no.: 112) • TEIF8 interrupt [channel 8] (group interrupt source no.: 0) • ERIF8 interrupt [channel 8] (group interrupt source no.: 1) • BRIF8 interrupt [channel 8] (group interrupt source no.: 2) • DRIF8 interrupt [channel 8] (group interrupt source no.: 3) • TEIF9 interrupt [channel 9] (group interrupt source no.: 4) • ERIF9 interrupt [channel 9] (group interrupt source no.: 5) • BRIF9 interrupt [channel 9] (group interrupt source no.: 6) • DRIF9 interrupt [channel 9] (group interrupt source no.: 7) • TEIF10 interrupt [channel 10] (group interrupt source no.: 8) • ERIF10 interrupt [channel 10] (group interrupt source no.: 9) • BRIF10 interrupt [channel 10] (group interrupt source no.: 10) • DRIF10 interrupt [channel 10] (group interrupt source no.: 11) • TEIF11 interrupt [channel 11] (group interrupt source no.: 12) • ERIF11 interrupt [channel 11] (group interrupt source no.: 13) • BRIF11 interrupt [channel 11] (group interrupt source no.: 14) • DRIF11 interrupt [channel 11] (group interrupt source no.: 15)

2.6 Header Files

All API calls and their supporting interface definitions are located in r_scif_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_scif_rx_config.h. The option names and setting values are listed in the table below:

Configuration options in r_scif_rx_config.h	
`#define SCIF_CFG_PARAM_CHECKING_ENABLE 1`	If this equate is set to 1, parameter checking is included in the build. If the equate is set to 0, the parameter checking is omitted from the build. Setting this equate to BSP_CFG_PARAM_CHECKING_ENABLE utilizes the system default setting.
`#define SCIF_CFG_ASYNC_INCLUDED 1` `#define SCIF_CFG_SYNC_INCLUDED 0`	These equates 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 SCIF_CFG_CH8_INCLUDED 0`
`#define SCIF_CFG_CH9_INCLUDED 1`
`#define SCIF_CFG_CH10_INCLUDED 0`
`#define SCIF_CFG_CH11_INCLUDED 0`
	Each channel has associated with it transmit and receive pointers, counters, interrupts, and other program and RAM resources. Setting a #define to 1 allocates resources for that channel.

`#define SCIF_CFG_CH8_TX_FIFO_THRESHOLD 8`
`#define SCIF_CFG_CH9_TX_FIFO_THRESHOLD 8`
`#define SCIF_CFG_CH10_TX_FIFO_THRESHOLD 8`
`#define SCIF_CFG_CH11_TX_FIFO_THRESHOLD 8`
	The transmit FIFO is 16 bytes deep. A TXIF interrupt occurs when there are threshold number of bytes remaining in the FIFO, indicating it is time to load more bytes. Valid values are 0 through 15. Ideally, all messages sent are multiples of the threshold value, and the threshold value is small enough such that no gaps occur between bytes during transmission at high bit rates due to reloading the FIFO.

`#define SCIF_CFG_CH8_RX_FIFO_THRESHOLD 8`
`#define SCIF_CFG_CH9_RX_FIFO_THRESHOLD 8`
`#define SCIF_CFG_CH10_RX_FIFO_THRESHOLD 8`
`#define SCIF_CFG_CH11_RX_FIFO_THRESHOLD 8`
	The receive FIFO is 16 bytes deep. An RXIF interrupt occurs when there are threshold number of bytes available in the FIFO, indicating it is time to read more bytes. Valid values are 1 through 16. Ideally, all messages received are multiples of the threshold value, and the threshold value is small enough such that no overflow occurs while receiving at high bit rates due to insufficient time to read the FIFO.
	In Synchronous mode, these values should match the corresponding TX_FIFO_THRESHOLD for maximum efficiency.

2.9 Code Size

The sizes of ROM, RAM and maximum stack usage associated with this module and BSP 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 values in the table below are confirmed under the following conditions.

Module Revision: r_scif_rx rev2.00, r_bsp rev5.50

Compiler Version: Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00

(The option of "-lang = c99" is added to the default settings of the integrated development environment.)

GCC for Renesas RX 8.3.0.201904

(The option of "-std=gnu99" is added to the default settings of the integrated development environment.)

IAR C/C++ Compiler for Renesas RX version 4.12.1

(The default settings of the integrated development environment.)

Configuration Options: Default settings

ROM, RAM and Stack Code Sizes for Renesas Compiler				
Device / Communication methods / Number of channels	Category	Memory Used		
		ROM	RAM	STACK*1
RX64M Async only 1 channel	With Parameter Checking	13,372 bytes	7,769 bytes	220 bytes
	Without Parameter Checking	13,071 bytes		
RX64M Async only 2 channels	With Parameter Checking	13,551 bytes	7,817 bytes	220 bytes
	Without Parameter Checking	13,250 bytes		
RX64M Sync only 1 channel	With Parameter Checking	12,225 bytes	7,764 bytes	208 bytes
	Without Parameter Checking	11,995 bytes		
RX64M Sync only 2 channels	With Parameter Checking	12,366 bytes	7,812 bytes	208 bytes
	Without Parameter Checking	12,136 bytes		
RX64M Async only 1 channel Sync only 1 channel	With Parameter Checking	13,886 bytes	7,817 bytes	220 bytes
	Without Parameter Checking	13,535 bytes		

Note 1. The sizes of maximum usage stack of Interrupts functions are included.

ROM, RAM and Stack Code Sizes for GNU Compiler				
Device / Communication methods / Number of channels	Category	Memory Used		
		ROM	RAM	STACK*1
RX64M Async only 1 channel	With Parameter Checking	23,464 bytes	7,588 bytes	-
	Without Parameter Checking	22,880 bytes		
RX64M Async only 2 channels	With Parameter Checking	23,648 bytes	7,636 bytes	-
	Without Parameter Checking	23,064 bytes		
RX64M Sync only 1 channel	With Parameter Checking	21,424 bytes	7,584 bytes	-
	Without Parameter Checking	21,032 bytes		
RX64M Sync only 2 channels	With Parameter Checking	21,528 bytes	7,632 bytes	-
	Without Parameter Checking	21,144 bytes		
RX64M Async only 1 channel Sync only 1 channel	With Parameter Checking	24,096 bytes	7,588 bytes	-
	Without Parameter Checking	23,432 bytes		

Note 1. The sizes of maximum usage stack of Interrupts functions are included.

ROM, RAM and Stack Code Sizes for IAR Compiler				
Device / Communication methods / Number of channels	Category	Memory Used		
		ROM	RAM	STACK*1
RX64M Async only 1 channel	With Parameter Checking	14,990 bytes	5,267 bytes	244 bytes
	Without Parameter Checking	14,519 bytes		
RX64M Async only 2 channels	With Parameter Checking	15,131 bytes	5,315 bytes	244 bytes
	Without Parameter Checking	14,657 bytes		
RX64M Sync only 1 channel	With Parameter Checking	13,570 bytes	5,266 bytes	240 bytes
	Without Parameter Checking	13,290 bytes		
RX64M Sync only 2 channels	With Parameter Checking	13,673 bytes	5,314 bytes	240 bytes
	Without Parameter Checking	13,393 bytes		
RX64M Async only 1 channel Sync only 1 channel	With Parameter Checking	15,499 bytes	5,315 bytes	244 bytes
	Without Parameter Checking	14,993 bytes		

Note 1. The sizes of maximum usage stack of Interrupts functions are included.

2.10 Parameters

The API data structures are located in the file “r_scif_rx_if.h” and discussed in Section 3.

2.11 Return Values

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

```
typedef enum e_scif_err       // SCIF API error codes
{
    SCIF_SUCCESS=0,
    SCIF_ERR_BAD_CHAN,        // non-existent channel number
    SCIF_ERR_OMITTED_CHAN,    // SCI_CHx_INCLUDED is 0 in config.h
    SCIF_ERR_CH_NOT_CLOSED,   // channel still running in another mode
    SCIF_ERR_BAD_MODE,        // unsupported mode for channel
    SCIF_ERR_INVALID_ARG,     // argument is not valid for parameter
    SCIF_ERR_NULL_PTR,        // received null ptr; missing required argument
    SCIF_ERR_BUSY,            // 2 requests already being processed
    SCIF_ERR_IN_PROGRESS,     // request still being processed
} scif_err_t;
```

2.12 Callback Function

The callback function is specified by storing the address of the user function in the argument (p_callback) of the R_SCIF_Open() function.
For details on callback functions, refer to the R_SCIF_Open() function in Secion 3.

2.13 Adding the FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends using "Smart Configurator" described in (1) or (3). However, "Smart Configurator" only supports some RX devices. Please use the methods of (2) or (4) for unsupported RX devices.

(1) Adding the FIT module to your project using "Smart Configurator" in e² studio
By using the "Smart Configurator" in e² studio, the FIT module is automatically added to your project. Refer to "Renesas e² studio Smart Configurator User Guide (R20AN0451)" for details.

(2) Adding the FIT module to your project using "FIT Configurator" in e² studio
By using the "FIT Configurator" in e² studio, the FIT module is automatically added to your project. Refer to "Adding Firmware Integration Technology Modules to Projects (R01AN1723)" for details.

(3) Adding the FIT module to your project using "Smart Configurator" on CS+
By using the "Smart Configurator Standalone version" in CS+, the FIT module is automatically added to your project. Refer to "Renesas e² studio Smart Configurator User Guide (R20AN0451)" for details.

(4) 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".

Target devices describing "WAIT_LOOP"

- RX64M, RX71M Group

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 */
```

2.15 Limitations

2.15.1 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 CCRX project (e2 studio V7.5.0), the RAM start address is set as 0x4 to prevent the variable from being located at address 0x0. In the case of GCC project (e2 studio V7.5.0) and IAR project (EWRX V4.12.1), the start address of RAM is 0x0, so the above measures are necessary.

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

3. API Functions

R_SCIF_Open()

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

Format

```
scif_err_t  R_SCIF_Open(
   uint8_t const                  chan,
   scif_mode_t  const             mode,
   scif_cfg_t * const             p_cfg,
   void    (* const p_callback)(void *p_args),
   scif_hdl_t * const             p_hdl
);
```

Parameters

uint8_t *chan*
Channel to initialize; 8-11

scif_mode_t const *mode*
Operational mode (see enumeration below)

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

void (const p_callback)(void *p_args)*
Optional pointer to function called from interrupt when a message send/receive completes or receiver error occurs.

*scif_hdl_t * const* *p_hdl*
Pointer to a handle for channel (value set here)

The following SCIF 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_scif_mode    // SCIF operational modes
{
    SCIF_MODE_OFF=0,        // channel not in use
    SCIF_MODE_ASYNC,        // Asynchronous
    SCIF_MODE_SYNC,         // Synchronous
    SCIF_MODE_END_ENUM
} scif_mode_t;
```

The following enumerations indicate configurable options for Asynchronous mode used in its configuration structure. These values correspond to bit definitions in the SCR and SMR registers.

```
typedef enum e_scif_clk
{
    SCIF_CLK_INT    = 0x00,  // use internal clock for baud generation
    SCIF_CLK_EXT8X  = 0x03,  // use external clock 8x baud rate
    SCIF_CLK_EXT16X = 0x02   // use external clock 16x baud rate
} scif_clk_t;
```

```
typedef enum e_scif_size
{
    SCIF_DATA_7BIT = 0x40,
    SCIF_DATA_8BIT = 0x00
} scif_size_t;
```

```
typedef enum e_scif_parity_en
{
    SCIF_PARITY_ON  = 0x20,
    SCIF_PARITY_OFF = 0x00
} scif_parity_en_t;
```

```
typedef enum e_parity_t
{
    SCIF_ODD_PARITY  = 0x10,
    SCIF_EVEN_PARITY = 0x00
} scif_parity_t;
```

```
typedef enum e_scif_stop_t
{
    SCIF_STOPBITS_2 = 0x08,
    SCIF_STOPBITS_1 = 0x00
} scif_stop_t;
```

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_scif_uart
{
    uint32_t          baud_rate;         // ie 9600, 19200, 115200
    scif_clk_t        clk_src;
    scif_size_t       data_size;
    scif_parity_en_t  parity_en;
    scif_parity_t     parity_type;
    scif_stop_t       stop_bits;
    uint8_t           txif_priority;     // txif INT priority; 1=low, 15=high
    uint8_t           rxif_priority;     // rxif INT priority; 1=low, 15=high
    uint8_t           group_priority;    // teif, erif, brif INT priority;
                                         // must be greater than rx_priority
} scif_uart_t;
```

The configuration structure for Synchronous mode is as follows:

```
typedef struct st_scif_sync
{
    uint32_t    bit_rate;        // ie 1000000 for 1Mbps
    bool        msb_first;
```

```
    uint8_t     int_priority;    // transceiver interrupt priority; 1=low,
15=high
} scif_sync_t;
```

The union for p_cfg is:

```
typedef union
{
    scif_uart_t     async;
    scif_sync_t     sync;
} scif_cfg_t;
```

Return Values

SCIF_SUCCESS:	*Successful; channel initialized*
SCIF_ERR_BAD_CHAN:	*Channel number is invalid for part*
SCIF_ERR_OMITTED_CHAN:	*Corresponding SCIF_CHx_INCLUDED is 0*
SCIF_ERR_CH_NOT_CLOSED:	*Channel currently in operation; Perform R_SCIF_Close() first*
SCIF_ERR_BAD_MODE:	*Specified mode not currently supported*
SCIF_ERR_NULL_PTR:	*p_cfg or p_hdl pointer is NULL*
SCIF_ERR_INVALID_ARG:	*An element of the p_cfg structure contains an invalid value.*

Properties

Prototyped in file “r_scif_rx_if.h”

Description

Initializes an SCIF channel for a particular mode and provides a handle in **p_hdl* for use with other API functions. All applicable interrupts are enabled.

Example: Asynchronous Mode

```
scif_cfg_t   config;
scif_hdl_t   Console;
scif_err_t   err;

config.async.baud_rate = 115200;
config.async.clk_src = SCIF_CLK_INT;          // use internal clock
config.async.data_size = SCIF_DATA_8BIT;
config.async.parity_en = SCIF_PARITY_OFF;
config.async.parity_type = SCIF_EVEN_PARITY; // ignored (parity is disabled)
config.async.stop_bits = SCIF_STOPBITS_1;
config.async.tx_priority = 2;
config.async.rx_priority = 2;
config.async.rx_err_priority = 3;            // must be higher than rx_priority

err = R_SCIF_Open(SCI_CH9, SCI_MODE_ASYNC, &config, MyCallback, &Console);
```

Example: Synchronous Mode

```
scif_cfg_t   config;
scif_hdl_t   syncHandle;
scif_err_t   err;

config.sync.bit_rate = 1000000;            // 1 Mbps
config.sync.msb_first = true;
config.sync.int_priority = 4;
err = R_SCIF_Open(SCI_CH8, SCI_MODE_SYNC, &config, syncCallback, &syncHandle);
```

Special Notes:

The driver uses an algorithm for calculating the optimum values for BRR, MDDR, SEMR.ABCS0, SEMR.BGDM and SMR.CKS using BSP_PCLKA_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.

The application must wait one bit-time after calling Open() before sending/receiving to allow the clock to settle.

If an external clock is used in Asynchronous mode, the Pin Function Select and port pins must be initialized first. The following is an example initialization for channel 9:

```
    MPC.PB5PFS.BYTE = 0x0A;    // Pin Func Select PB5 SCK9; clock as input
    PORTB.PDR.BIT.B5 = 0;      // set SCK pin direction to input (dflt)
    PORTB.PMR.BIT.B5 = 1;      // set SCK pin mode to peripheral
```

For initializing the clock in synchronous mode for channel 9:

```
    MPC.PB5PFS.BYTE = 0x0A;    // Pin Func Select PB5 SCK9; clock as output
    PORTB.PDR.BIT.B5 = 1;      // set SCK pin direction to output
    PORTB.PMR.BIT.B5 = 1;      // set SCK pin mode to peripheral
```

The callback function has a single argument. This is a pointer to a structure which is cast to a void pointer (provides consistency with other FIT module callback functions). The structure is as follows:

```
typedef struct st_scif_cb_args // callback arguments
{
    scif_hdl_t       hdl;
    scif_cb_evt_t    event;
} scif_cb_args_t;
```

The “hdl” argument is the handle for the channel. The possible events passed are defined in the following enumeration:

```
typedef enum e_scif_cb_evt    // callback function events
{
    // Async Events
    SCIF_EVT_TX_DONE,         // Send() requests processed; last bit transmitted
    SCIF_EVT_RX_DONE,         // Receive() request processed;
                              // some or no data may be in RX FIFO
    SCIF_EVT_RX_BREAK,        // received BREAK condition
    SCIF_EVT_RX_OVERFLOW,     // receiver FIFO overrun error
    SCIF_EVT_RX_FRAMING_ERR,  // received framing error
    SCIF_EVT_RX_PARITY_ERR,   // received parity error

    // Sync Events
    SCIF_EVT_XCV_DONE,        // All requests processed
    SCIF_EVT_XCV_ABORTED      // transfer aborted; FIFOs flushed
} scif_cb_evt_t;
```

The events SCIF_EVT_FRAMING_ERR and SCIF_EVT_PARITY_ERR indicate that the next byte to be read from the FIFO has an error. This byte is not passed to the callback function but is loaded into the receive buffer. This is so the bytes read from the FIFO will match the requested count. An example template for an Asynchronous mode callback function is provided here:

```
void MyCallback(void *p_args)
{
scif_cb_args_t    *args;


    args = (scif_cb_args_t *)p_args;

    switch (args->event)
    {
    case SCIF_EVT_TX_DONE:
        // from TEIF interrupt; all data sent
        nop();
        break;

    case SCIF_EVT_RX_DONE:
        // from final RXIF interrupt; all requested bytes have been received
        // some or no data may be in RX FIFO
        nop();
        break;

    case SCIF_EVT_RX_BREAK:
        // from BRIF interrupt; received BREAK condition
        // error condition is cleared in BRIF routine
        nop();
        break;

    case SCIF_EVT_RX_OVERFLOW:
        // from BRIF interrupt; receiver overrun error occurred
        // error condition is cleared in BRIF routine
        nop();
        break;

    case SCIF_EVT_RX_FRAMING_ERR:
        // from ERIF interrupt; receiver framing error occurred
        // error condition is cleared in ERIF routine
        nop();
        break;

    case SCIF_EVT_RX_PARITY_ERR:
        // from ERIF interrupt; receiver parity error occurred
        // error condition is cleared in ERIF routine
        nop();
        break;
    };

}
```

An example template for a Synchronous mode callback function is provided here:

```
void syncCallback(void *p_args)
{
scif_cb_args_t    *args;


    args = (scif_cb_args_t *)p_args;

    if (args->event == SCIF_EVT_XCV_DONE)
    {
        // from TEIF interrupt; all data sent
        // data transfer request(s) completed
        nop();
    }
    else if (args->event == SCIF_EVT_XCV_ABORTED)
    {
        // data transfer aborted
        nop();
    }
}
```

R_SCIF_Close()

This function removes power to the SCIF channel and disables the associated interrupts.

Format

```
scif_err_t   R_SCIF_Close(
   scif_hdl_t const   hdl
);
```

Parameters

hdl
Handle for channel

Return Values

SCIF_SUCCESS:	*Successful; channel closed*
SCIF_ERR_NULL_PTR:	*hdl is NULL*

Properties

Prototyped in file “r_scif_rx_if.h”

Description

Disables the SCIF 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.

Example

```
scif_hdl_t   Console;

err = R_SCIF_Open(SCI_CH9, SCI_MODE_ASYNC, &config, MyCallback, &Console);

err = R_SCIF_Close(Console);
```

Special Notes:

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

R_SCIF_Send()

Queues up to two requests. Begins transmission if transmitter is not already in use.

Format

```
scif_err_t   R_SCIF_Send(
   scif_hdl_t const   hdl,
   uint8_t            *p_src,
   uint16_t const     length
);
```

Parameters

scif_hdl_t *hdl*
Handle for channel

uint8_t *p_src*
Pointer to data to transmit

uint16_t *length*
Number of bytes to send

Return Values

SCIF_SUCCESS:	*Message queued for sending; transmission started if transmitter is idle.*
SCIF_ERR_NULL_PTR:	*hdl or p_src is NULL*
SCIF_ERR_BAD_MODE:	*Channel mode not currently supported*
SCIF_ERR_INVALID_ARG:	*length is 0*
SCIF_ERR_BUSY:	*Cannot process request. 2+ requests already placed*

Properties

Prototyped in file “r_scif_rx_if.h”

Description

If the driver can process the request, SCIF_SUCCESS is returned. If there are already two requests outstanding, SCIF_ERR_BUSY is returned. If a message is longer than the FIFO size, the driver will automatically reload the FIFO at the interrupt level each time the threshold level (set in config.h) is reached.

When no more data remains to be transmitted, an SCIF_EVT_TX_DONE (Async) or SCIF_EVT_XCV_DONE (Sync) event is passed to the callback function if specified in Open(). If no callback function was provided, the application must poll for completion using a Control() command.

If it is desired to know when each message has completed transmission, do not have more than one Send() request outstanding at a time. This driver is optimized for streaming data and the “done” event is used to indicate transmit completion of all data.

Example 1: Asynchronous Mode Blocking

```
uint8_t     g_data_block[128];

scif_cfg_t  config;
scif_hdl_t  hdl;
scif_err_t  err;

err = R_SCIF_Open(SCI_CH9, SCI_MODE_ASYNC, &config, NULL, &hdl);
  :

/* Check if transmitter available (and wait if necessary) to send message */
while (R_SCIF_Send(hdl, g_data_block, 128) == SCIF_ERR_TX_BUSY)
{
    /* wait until a send request can be queued */
}

/* Block for message to complete sending */
while (R_SCIF_Control(hdl,SCIF_CMD_CHECK_TX_DONE, NULL) == SCIF_ERR_IN_PROGRESS)
{
    /* do other processing if desired while waiting for send to complete */
}
```

Example 2: Asynchronous Mode Non-Blocking

```
uint8_t     g_data_block[128];

scif_cfg_t  config;
scif_hdl_t  hdl;
scif_err_t  err;

err = R_SCIF_Open(SCI_CH9, SCI_MODE_ASYNC, &config, MyCallback, &hdl);
  :

/* if know 1 or no requests outstanding, can issue Send() immediately  */
R_SCIF_Send(hdl, g_data_block, 128);



void MyCallback(void *p_args)
{
scif_cb_args_t    *args;

    args = (scif_cb_args_t *)p_args;
    switch (args->event)
    {
    case SCIF_EVT_TX_DONE:
        // all data successfully sent
        break;

    case SCIF_EVT_RX_BREAK:
        // received break; handle error condition
        R_SCIF_Control(args->hdl, SCIF_CMD_RESET_TX, NULL);
        R_SCIF_Control(args->hdl, SCIF_CMD_RESET_RX, NULL);
        break;
    };
}
```

Example 3: Synchronous Mode Blocking

```
#define STRING    “Test String”

scif_cfg_t  config;
scif_hdl_t  lcdHandle;
scif_err_t  err;

err = R_SCIF_Open(SCI_CH8, SCI_MODE_SYNC, &config, NULL, &lcdHandle);
  :

/* Check if transmitter available (and wait if necessary) to send message */
while (R_SCIF_Send(lcdHandle, STRING1, sizeof(STRING1)) == SCIF_ERR_BUSY)
{
    /* wait until a send request can be queued */
}

/* Block for message to complete sending */
while(R_SCIF_Control(lcdHandle,SCIF_CMD_CHECK_XCV_DONE,NULL) == SCIF_ERR_IN_PROGRESS)
{
    /* do other processing if desired while waiting for send to complete */
}
```

Example 4: Synchronous Mode Non-Blocking

```
#define STRING    “Test String”

scif_cfg_t  config;
scif_hdl_t  lcdHandle;
scif_err_t  err;

err = R_SCIF_Open(SCI_CH8, SCI_MODE_SYNC, &config, syncCallback, &lcdHandle);
  :

/* if know 1 or no requests outstanding, can issue Send() immediately  */
R_SCIF_Send(lcdHandle, STRING1, sizeof(STRING1));


void syncCallback(void *p_args)
{
scif_cb_args_t    *args;


    args = (scif_cb_args_t *)p_args;

    if (args->event == SCIF_EVT_XCV_DONE)
    {
        // data transfer completed; do any processing here
        // nop();
    }
    else if (args->event == SCIF_EVT_XCV_ABORTED)
    {
        // data transfer aborted; do any processing here
    }
}
```

Special Notes:

In synchronous mode, the peripheral drives the clock for Send(), Receive(), and SendReceive() messages. In this mode, at most two transfer requests of any kind can ever be outstanding at a time. Therefore a SCIF_ERR_BUSY may be returned even when no Send() message was previously issued.

Do not re-use the same buffer pointed to by *p_src* until it is known that the previous message the buffer was used for has completed transmission. Doing so could corrupt the data of the message currently being sent. This behavior is different than the standard SCI driver which copies the original buffer into a queue where it waited until it could be transmitted. For high throughput, this driver does not copy data into an intermediate queue and the hardware FIFO is the only temporary storage mechanism.

R_SCIF_Receive()

Queues up to two requests. Fetches data from the hardware FIFO. In Synchronous mode, initiates clocking of data if not already in use.

Format

```
scif_err_t   R_SCIF_Receive(
   scif_hdl_t const       hdl,
   uint8_t                *p_dst,
   uint16_t const         length
);
```

Parameters

scif_hdl_t const *hdl*
Handle for channel

uint8_t **p_dst*
Pointer to buffer to load data into

uint16_t const *length*
Number of bytes to read

Return Values

SCIF_SUCCESS:	*Request queued. Clocking begins (Sync) if transceiver idle*
SCIF_ERR_NULL_PTR:	*hdl value is NULL*
SCIF_ERR_BAD_MODE:	*Channel mode not currently supported*
SCIF_ERR_INVALID_ARG:	*length is 0*
SCIF_ERR_BUSY:	*Cannot process request. 2+ requests already placed*

Properties

Prototyped in file "r_scif_rx_if.h"

Description

If the driver can process the request, SCIF_SUCCESS is returned. If there are already two requests outstanding, SCIF_ERR_BUSY is returned. If a message is longer than the FIFO size, the driver will automatically read from the FIFO at the interrupt level each time the threshold level (set in config.h) is reached. If there is less than the threshold level bytes remaining the driver automatically adjusts the threshold level.

When no more data remains to be received, an SCIF_EVT_RX_DONE (Async) or SCIF_EVT_XCV_DONE (Sync) event is passed to the callback function if specified in Open(). If no callback function was provided, the application must poll for completion using a Control() command. Note that errors which occurred during reception are only reported via the callback function.

If it is desired to know when each message has completed reception, do not have more than one Receive() request outstanding at a time. This driver is optimized for streaming data and the "done" event is used to indicate receive completion of all requested data.

Example 1: Asynchronous Blocking

```
uint8_t     g_data_block[128];

scif_cfg_t  config;
scif_hdl_t  hdl;
scif_err_t  err;

err = R_SCIF_Open(SCI_CH9, SCI_MODE_ASYNC, &config, NULL, &hdl);
  :

/* Check if receiver available (and wait if necessary) to receive message */
while (R_SCIF_Receive(hdl, g_data_block, 128) == SCIF_ERR_RX_BUSY)
{
    /* wait until a receive request can be queued */
}

/* Block for request to complete */
while (R_SCIF_Control(hdl,SCIF_CMD_CHECK_RX_DONE,NULL) == SCIF_ERR_IN_PROGRESS)
{
    /* do other processing if desired while waiting for receive to complete */
}
```

Example 2: Asynchronous Non-Blocking

```
uint8_t     g_data[8];

scif_cfg_t  config;
scif_hdl_t  hdl;
scif_err_t  err;

err = R_SCIF_Open(SCI_CH9, SCI_MODE_ASYNC, &config, MyCallback, &hdl);
  :

/* Check if receiver available (and wait if necessary) to receive message.
 * Don't block for request to complete.
 */
while (R_SCIF_Receive(hdl, g_data, 8) == SCIF_ERR_RX_BUSY);
{
    /* wait until receive request can be queued */
}


/* An example of processing receive events */

void MyCallback(void *p_args)
{
scif_cb_args_t  *args;
static bool      err_flg=false;
uint8_t          byte;

    args = (scif_cb_args_t *)p_args;
    switch (args->event)
    {
    case SCIF_EVT_RX_FRAMING_ERR:
    case SCIF_EVT_RX_PARITY_ERR:
        /* Continue to receive msg, but set flag to indicate error detected */
        err_flg = true;
        break;

    case SCIF_EVT_RX_OVERFLOW:
        /* Overrun occurred. Issue "abort" to sender and reset err_flg to start
         * fresh. Driver automatically resets FIFOs when break is generated.
```

```
         */
        R_SCIF_Control(args->hdl, SCIF_CMD_GENERATE_BREAK, NULL);
        err_flg = false;
        break;

    case SCIF_EVT_RX_BREAK:
        /* Received break. Reset transmitter, receiver, and err_flg.
        R_SCIF_Control(args->hdl, SCIF_CMD_RESET_TX, NULL);
        R_SCIF_Control(args->hdl, SCIF_CMD_RESET_RX, NULL);
        err_flg = false;
        break;

    case SCIF_EVT_RX_DONE:
        /* Done receiving message. Issue ACK or NAK based upon err_flg. */
        byte = (err_flg == true) ? NAK : ACK;
        R_SCIF_Send(hdl, &byte,1);
        err_flg = false;
        break;
    };
}
```

Example 3: Synchronous Mode Blocking

```
uint8_t     g_block[2][128];

scif_cfg_t  config;
scif_hdl_t  hdl;
scif_err_t  err;

err = R_SCIF_Open(SCI_CH9, SCI_MODE_SYNC, &config, NULL, &hdl);
  :

/* Issue two Receive() calls and wait for completion */
while (R_SCIF_Receive(hdl, &g_data_block[0], 128) == SCIF_ERR_XCV_BUSY)
{
    /* wait until receive request can be queued */
}

while (R_SCIF_Receive(hdl, &g_data_block[1], 128) == SCIF_ERR_XCV_BUSY)
{
    /* wait until receive request can be queued */
}

// (could replace above requests with single request for 256 with first address)

while (R_SCIF_Control(hdl,SCIF_CMD_CHECK_XCV_DONE, NULL) == SCIF_ERR_IN_PROGRESS)
{
    /* do other processing if desired while waiting for receive to complete */
}
```

Example 4: Synchronous Mode Non-Blocking

```
uint8_t      sensor_cmd,sync_buf[10];
scif_cfg_t  config;
scif_hdl_t  hdl;
scif_err_t  err;

err = R_SCIF_Open(SCI_CH9, SCI_MODE_SYNC, &config, syncCallback, &hdl);


/* SEND COMMAND TO SENSOR TO PROVIDE CURRENT READING AND GET DATA */

sensor_cmd = SNS_CMD_READ_LEVEL;

/* FIFOs known to be empty here; can have two outstanding msg requests */
R_SCIF_Send(hdl, &sensor_cmd, 1);
R_SCIF_Receive(hdl, sync_buf, 4);

/* do not wait for reply */
```

Special Notes:

In synchronous mode, the peripheral drives the clock for Send(), Receive(), and SendReceive() messages. In this mode, at most two transfer requests of any kind can ever be outstanding at a time. Therefore a SCIF_ERR_BUSY may be returned even when no Receive() message was previously issued.

Do not re-use the same buffer pointed to by *p_dst* until it is known that the previous message the buffer was used for has been processed. Doing so could corrupt the data of the message previously received.

R_SCIF_SendReceive()

For Synchronous mode only. Transmits and receives data simultaneously.

Format

```
scif_err_t   R_SCIF_SendReceive(
   scif_hdl_t const    hdl,
   uint8_t             *p_src,
   uint8_t             *p_dst,
   uint16_t const      length
);
```

Parameters

scif_hdl_t const *hdl*

Handle for channel

uint8_t **p_src*

Pointer to data to transmit

uint8_t **p_dst*

Pointer to buffer to load data into

uint16_t *length*

Number of bytes to send

Return Values

SCIF_SUCCESS:	*Data transfer queued and initiated if transceiver idle*
SCIF_ERR_NULL_PTR:	*hdl value is NULL*
SCIF_ERR_BAD_MODE:	*Channel mode not Synchronous*
SCIF_ERR_INVALID_ARG:	*length is 0*
SCIF_ERR_BUSY:	*Cannot process request. 2+ requests already placed*

Properties

Prototyped in file “r_scif_rx_if.h”

Description

This function transmits and receives data simultaneously if the transceiver is not in use. If the driver can process the request, SCIF_SUCCESS is returned. If there are already two requests outstanding, SCIF_ERR_BUSY is returned. If a message is longer than the FIFO size, the driver will automatically process the FIFO at the interrupt level each time the threshold level (set in config.h) is reached.

When no more data remains to be transmitted and received, an SCIF_EVT_XCV_DONE event is passed to the callback function if specified in Open(). If no callback function was provided, the application must poll for completion using a Control() command.

If it is desired to know when each message has completed transmission/reception, do not have more than one SendReceive() request outstanding at a time. This driver is optimized for streaming data and the “done” event is used to indicate transmit/receive completion of all data.

Example: Blocking

```
scif_hdl_t  hdl;
scif_err_t  err;
uint8_t     out_buf[2] = {SF_CMD_READ_STATUS_REG, SCIF_CFG_DUMMY_TX_BYTE };
uint8_t     in_buf[2] = {0x55, 0x55};           // init to illegal values
  :


/* Clock two bytes of data. The first byte is a command out (ignore byte in)
 * and the second byte is a response in (dummy byte clocked out)
 */

/* FIFOs known to be empty here */
R_SCIF_SendReceive(hdl, out_buf, in_buf, 2);

while (R_SCIF_Control(hdl, SCI_CMD_CHECK_XCV_DONE, NULL) == SCIF_ERR_BUSY)
{
    /* wait for completion */
}

// reply is in in_buf[1]
```

Special Notes:

In synchronous mode, the peripheral drives the clock for Send(), Receive(), and SendReceive() messages. In this mode, at most two transfer requests of any kind can ever be outstanding at a time. Therefore a SCIF_ERR_BUSY may be returned even when no SendReceive() message was previously issued.

Do not re-use the same buffers pointed to by *p_dst* and *p_dst* until it is known that the previous message the buffer was used for has been processed. Doing so could corrupt the data of the message previously received.

R_SCIF_Control()

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

Format

```
scif_err_t   R_SCIF_Control(
   scif_hdl_t const       hdl,
   scif_cmd_t const       cmd,
   void                   *p_args
);
```

Parameters

scif_hdl_t const *hdl*
Handle for channel

scif_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_scif_cmd         // SCIF Control() commands
{
    // Both modes
    SCIF_CMD_CHANGE_BAUD,        // change baud/bit rate

    // Async commands
    SCIF_CMD_EN_FLOW_CTRL,       // enable CTS/RTS flow control
    SCIF_CMD_EN_NOISE_CANCEL,    // enable noise cancellation
    SCIF_CMD_EN_MSB_FIRST,       // transmit/receive MSB first
    SCIF_CMD_GENERATE_BREAK,     // generate break condition; resets FIFOs
    SCIF_CMD_TX_BYTES_REMAINING, // number total bytes yet to transmit
    SCIF_CMD_RX_BYTES_PENDING,   // number bytes yet to receive
    SCIF_CMD_CHECK_TX_DONE,  // see if tx requests complete; SCIF_SUCCESS if yes
    SCIF_CMD_CHECK_RX_DONE,  // see if rx request complete; SCIF_SUCCESS if yes
    SCIF_CMD_RESET_TX,           // abort transmit requests; reset transmit FIFO
    SCIF_CMD_RESET_RX,           // abort receive requests; reset receive FIFO

    // Sync commands
    SCIF_CMD_CHECK_XCV_DONE,     // see if Send, Receive, or SendReceive
                                 //   requests are done; SCIF_SUCCESS if yes
    SCIF_CMD_RESET_XCV           // abort transfer requests; reset FIFOs
} scif_cmd_t;
```

Most of the commands do not require arguments and take NULL or FIT_NO_PTR for p_args. The argument structure for SCIF_CMD_CHANGE_BAUD is shown below. Note that this command may not be used for Asynchronous mode when using an external clock.

```
typedef struct _sci_baud
{
    uint32_t    pclk;           // PCLKA speed; ie 120000000 (120 MHz)
    uint32_t    rate;           // ie 9600, 19200, 115200
} sci_baud_t;
```

The argument for SCIF_CMD_TX_BYTES_REMAINING and SCIF_CMD_RX_BYTES_PENDING is a pointer to a uint16_t variable to hold a count value.

The commands SCIF_CMD_CHECK_TX_DONE, SCIF_CMD_CHECK_RX_DONE, and SCIF_CMD_CHECK_XCV_DONE return SCIF_SUCCESS when all requests have been transmitted. Otherwise SCIF_ERR_IN_PROGRESS is returned.

Note: For SCIF_CMD_RESET_TX, if a message transmission is in progress it will be abort immediately. It will not wait until the current byte completes transmission. In this case, it is recommended to wait 1 byte-time before sending again to allow receiver to process likely framing error from last [partial] byte sent.

Return Values

SCIF_SUCCESS:	*Successful; channel initialized*
SCIF_ERR_NULL_PTR:	*hdl or p_args pointer is NULL (when required)*
SCIF_ERR_BAD_MODE:	*Channel mode not currently supported*
SCIF_ERR_INVALID_ARG:	*The cmd value or an element of p_args contains an invalid value.*

Properties

Prototyped in file “r_scif_rx_if.h”

Description

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

Example 1: Asynchronous

```
scif_hdl_t   Console;
scif_cfg_t   config;
scif_baud_t  baud;
scif_err_t   err;
uint16_t     cnt;
  :
R_SCIF_Open(SCI_CH9, SCIF_MODE_ASYNC, &config, MyCallback, &Console);
R_SCIF_Control(Console, SCIF_CMD_EN_NOISE_CANCEL, NULL);
R_SCIF_Control(Console, SCIF_CMD_EN_MSB_FIRST, NULL);
  :
/* reset baud rate due to low power mode clock switching */
baud.pclk = 8000000;      // 8MHz
baud.rate = 19200;
R_SCIF_Control(Console, SCIF_CMD_CHANGE_BAUD, &baud);
  :
/* after initiating a large transmit, see how many bytes remaining to send */
R_SCIF_Control(Console, SCIF_CMD_TX_BYTES_REMAINING, &cnt);
// for progress bar: (message size - cnt)/(message size) = % complete
  :
/* after initiating a large receive, see how many bytes left to receive */
R_SCIF_Control(Console, SCIF_CMD_RX_BYTES_PENDING, &cnt);
// for progress bar: (request size - cnt)/(request size) = % complete
  :
```

Example 2: Synchronous

```
scif_cfg_t   config;
scif_hdl_t   syncHandle;
scif_err_t   err;

config.sync.bit_rate = 1000000;            // 1 Mbps
config.sync.msb_first = true;
config.sync.int_priority = 4;
err = R_SCIF_Open(SCI_CH8, SCI_MODE_SYNC, &config, syncCallback, &syncHandle);
  :

// after starting a large message transfer, abort transfer
R_SCIF_Control(syncHandle, SCIF_CMD_RESET_XCV, NULL);
```

Special Notes:

Do not use the value loaded by SCIF_CMD_TX_BYTES_REMAINING to determine if a message is sent. There still may be bits in the shift register when this commands return a “0”.Use SCIF_CMD_TX_DONE for this purpose.

Wait one bit-time after performing a SCIF_CMD_CHANGE_BAUD for the clock to settle at the new speed. The bit time should be measured in terms of the slower bit rate.

Wait two bit-times after performing a SCIF_CMD_GENERATE_BREAK before resuming communications. Any Send() or Receive() calls made during this will get a SCIF_ERR_BUSY until the break completes. A break condition lasts 1.5 to 2.0 byte times.

The driver uses an algorithm for calculating the optimum values for BRR, MDDR, SEMR.ABCS0, SEMR.BGDM, SEMR.BRME, SEMR.MDDRS and SMR.CKS. This however does not guarantee a low bit error rate for all peripheral clock/baud rate combinations.

If the command SCIF_CMD_EN_FLOW_CTRL is to be used, the Pin Function Select and port pins must be configured first. The following is an example initialization for channel 9:

```
    MPC.PB4PFS.BYTE = 0x0B;      // Pin Func Select PB4 CTS
    PORTB.PDR.BIT.B4 = 0;        // set CTS pin direction to input
    PORTB.PMR.BIT.B4 = 1;        // set CTS pin mode to peripheral

    MPC.PB5PFS.BYTE = 0x0B;      // Pin Func Select PB5 RTS
    PORTB.PDR.BIT.B5 = 1;        // set RTS pin direction to output
    PORTB.PMR.BIT.B5 = 1;        // set RTS pin mode to peripheral
```

R_SCIF_GetVersion()

This function returns the driver version number at runtime.

Format

```
uint32_t  R_SCIF_GetVersion(
   void
);
```

Parameters

None

Return Values

Version number.

Properties

Prototyped in file “r_scif_rx_if.h”

Description

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

Example

```
uint32_t   version;
  :
version = R_SCIF_GetVersion();
```

Special Notes:

None

4. 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).

4.1 scif_demo_rskrx64m

This is a simple demo of the SCIF API (r_scif_rx) which communicates with a terminal over SCIF channel 8 connected to the USB Virtual COM port. The user is first prompted at the terminal to enter a character, at which point the SCIF module version is transmitted to the terminal over the SCIF channel. The demo then enters a continuous loop waiting for a character to be entered and then transmitting two 160 byte buffers of char data. When the data has been sent a summary of the number of bytes sent and number of TXIF interrupts required to transmit the data is sent/displayed at the terminal. This demonstrates how the SCIF_CFG_CHx_TX_FIFO_THRESHOLD configuration value in r_scif_r_config.h affects the number of interrupts required to process all 320 bytes of data using the SCIF Tx FIFO.

Setup and Execution

1. Ensure driver support for channel 8 is enabled in r_scif_rx_config.h:

   => #define SCIF_CFG_CH8_INCLUDED (1)

2. Prepare the RSKRX64M board:

   - Jumpers J12, J14, J16 and J18: OFF

   - Connect J12 pin 2 to J16 pin 3 using a jumper wire

   - Connect J14 pin 2 to J18 pin 3 using a jumper wire

   This connects the CH8 Tx/Rx signals to the Virtual COM USB Tx/Rx signals.

3. Connect the RSK board serial port to a PC serial port. For this demo the RSKRX64M serial to USB Virtual COM Interface is used. In this case, connect the USB port to a PC with the Renesas USB-serial device driver installed. The USB will enumerate on the PC as a virtual COM port. Note the COM port number.
4. Open a terminal emulation program on the PC, such as "Tera Term", and select the serial COM port assigned to the Virtual COM Interface. Configure the terminal serial settings to match the settings in this sample application: 115200 baud, 8-bit data, no parity, 1 stop bit, no flow control.
5. Build and download this sample application to the RSK board. Run the application with the debugger.
6. A prompt to "Enter a char>" should now be seen at the terminal indicating that the demo is running.

Boards Supported

RSKRX64M

4.2 scif_demo_rskrx71m

This is a simple demo of the SCIF API (r_scif_rx) which communicates with a terminal over SCIF channel 8 connected to the USB Virtual COM port. The user is first prompted at the terminal to enter a character, at which point the SCIF module version is transmitted to the terminal over the SCIF channel. The demo then enters a continuous loop waiting for a character to be entered and then transmitting two 160 byte buffers of char data. When the data has been sent a summary of the number of bytes sent and number of TXIF interrupts required to transmit the data is sent/displayed at the terminal. This demonstrates how the SCIF_CFG_CHx_TX_FIFO_THRESHOLD configuration value in r_scif_r_config.h affects the number of interrupts required to process all 320 bytes of data using the SCIF Tx FIFO.

Setup and Execution

1. Ensure driver support for channel 8 is enabled in r_scif_rx_config.h:

   => #define SCIF_CFG_CH8_INCLUDED (1)

2. Prepare the RSKRX71M board:

   - Jumpers J12, J14, J16 and J18: OFF

   - Connect J12 pin 2 to J16 pin 3 using a jumper wire

   - Connect J14 pin 2 to J18 pin 3 using a jumper wire

     This connects the CH8 Tx/Rx signals to the Virtual COM USB Tx/Rx signals.

3. Connect the RSK board serial port to a PC serial port. For this demo the RSKRX71M serial to USB Virtual COM Interface is used. In this case, connect the USB port to a PC with the Renesas USB-serial device driver installed. The USB will enumerate on the PC as a virtual COM port. Note the COM port number.
4. Open a terminal emulation program on the PC, such as "Tera Term", and select the serial COM port assigned to the Virtual COM Interface. Configure the terminal serial settings to match the settings in this sample application: 115200 baud, 8-bit data, no parity, 1 stop bit, no flow control.
5. Build and download this sample application to the RSK board. Run the application with the debugger.
6. A prompt to "Enter a char>" should now be seen at the terminal indicating that the demo is running.

Boards Supported

RSKRX71M

4.3 Adding the 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”.

4.4 Downloading Demo Projects

Demo projects are not included in the RX Driver Package. When using the demo project, the FIT module needs to be downloaded. To download the FIT module, right click on the required application note and select “Sample Code (download)” from the context menu in the Smart Brower >> Application Notes tab.

5. Pin Setting

To use the SCIF 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 before calling the R_SCIF_Open function.

When performing the pin setting in the e^2 studio, the Pin Setting feature of the FIT Configurator or 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 FIT Configurator or the Smart Configurator. Then pins are configured by calling the function defined in the source file. Refer to Table 4.1 for details.

Table 5.1 Function Output by the FIT Configurator

MCU Used	Option Selected	Function to be Output	Remarks
RX64M, RX71M	SCIF8	R_SCIF_PinSet_SCIF8()	When using SCIF8.
	SCIF9	R_SCIF_PinSet_SCIF9()	When using SCIF9.
	SCIF10	R_SCIF_PinSet_SCIF10()	When using SCIF10
	SCIF11	R_SCIF_PinSet_SCIF11()	When using SCIF11.

6. Appendices

6.1 Operating Test Environment

This section describes for detailed the operating test environments of this module.

Table 6.1 Operation Confirmation Environment for Rev.1.22.

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.3.0
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
Endian	Big endian/little endian
Revision of the module	Rev.1.22

Table 6.2 Operation Confirmation Environment for Rev.1.21

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.1.0
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.00.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
Endian	Big endian/little endian
Revision of the module	Rev.1.21

Table 6.3 Confirmed Operation Environment (Rev. 2.00)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.6.0 IAR Embedded Workbench for Renesas RX 4.12.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.201904 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 4.12.1 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 RX64M (product No.: R0K50564Mxxxxxx) Renesas Starter Kit+ for RX71M (product No.: R0K50571Mxxxxxx)

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:

- When using CS+:

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

- When using e² studio:

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

When using a FIT module, the board support package FIT module (BSP module) must also be added to the project. For this, 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_sci_iic_rx module.

A: The FIT module you added may not support the target device chosen in the user project. Check if the FIT module supports the target device for the project used.

(3) Q: I have added the FIT module to the project and built it. Then I got an error for when the configuration setting is wrong.

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

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Aug. 31, 14	—	Initial Release.
1.10	Mar. 19, 15	1,3,27,28	Added support for RX71M and RX64M/RX71M demos
1.20	Mar. 16, 17	—	Fixed bug that caused extra clocks to be sent at high speeds in SYNC mode.
1.21	Dec. 07, 18	1	Related Documents: Added the following document: “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)”
		4	2.3 Software Requirements: Revised.
			2.4 Limitations: Deleted.
		5	2.5 Interrupt Vector: Added.
		8	2.11. Adding the FIT Module to Your Project: Revised.
		29	4. Demo Projects: Revised.
		30	4.4 Downloading Demo Projects: Added.
		31	5.1. Confirmed Operation Environment: Added.
			5.2. Troubleshooting: Added.
		32	Related Technical Updates: Added.
		Program	Added document number of the application note accompanying the sample program of the FIT module to xml file.
1.22	Apr. 01, 19	—	Changes associated with functions: Added support setting function of configuration option Using GUI on Smart Configurator. [Description] Added a setting file to support configuration option setting function by GUI.
		1	Changed Introduction.
		4	Added 1.1 SCIF FIT Module.
		5	Moved 1.3 API Overview.
		6	Changed 2 API Information.
		8	Changed 2.6 Header Files. Changed 2.7 Integer Types. Changed 2.8 Configuration Overview.
		9	Changed 2.9 Code Size.
		10	Changed 2.10 Parameters. Changed 2.11 Return Values. Added 2.12 Callback Function.
		11	Changed 2.13 Adding the FIT Module to Your Project.
		12	Added 2.14 “for”, “while” and “do while” statements.
		36	Added 5. Pin Setting.
		37	6.1 Operation Confirmation Environment: Added table for Rev.1.22.
2.00	Nov. 01, 19	—	Supported the following compilers. - GCC for Renesas RX - IAR C/C++ Compiler for Renesas RX
		—	Updated Demo Projects.
		1	Fixed Related Documents.
		9	Updated 2.9 Code Size.
		14	Added 2.15 Limitations.

Rev.	Date	Description	
		Page	Summary
2.00	Nov. 01, 19	39	6.1 Operation Confirmation Environment: Added table for Rev.2.00.
		Program	Guarantee atomicity in the critical section of the following register control. - Module Stop Control (MSTPCR) - Interrupt Request Enable Control (IEN) - Group Interrupt Request Enable (GENBL)

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.

Renesas Electronics Corporation

SALES OFFICES http://www.renesas.com

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

Renesas Electronics Corporation
TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan

Renesas Electronics America Inc.
1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.
Tel: +1-408-432-8888, Fax: +1-408-434-5351

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

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 101-T01, Floor 1, Building 7, Yard No. 7, 8th Street, Shangdi, Haidian District, Beijing 100085, 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 200333, China
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 No 3A-1 Level 3A Tower 8 UOA Business Park, No 1 Jalan Pengaturcara U1/51A, Seksyen U1, 40150 Shah Alam, Selangor, Malaysia
Tel: +60-3-5022-1288, Fax: +60-3-5022-1290

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338

Colophon 8.0