User's Manual

Smart Configurator

User's Manual: RX API Reference

Target Device
RX Family

All information contained in these materials, including products and product specifications, represents information on the product at the time of publication and is subject to change by Renesas Electronics Corp. without notice. Please review the latest information published by Renesas Electronics Corp. through various means, including the Renesas Electronics Corp. website (http://www.renesas.com).

Renesas Electronics
www.renesas.com

Rev.1.10 Apr. 2025

How to Use This Manual

Readers	The target readers of this manual are the application system engineers who use the Code Generator and need to understand its function.	
Purpose	The purpose of this manual is to explain the user for understanding and using the Code Generator functions. We aim to help their system development including their hardware and software.	
Organization	This manual can be broadly divided into the following units. 1.GENERAL 2.OUTPUT FILES 3.API FUNCITONS	
How to Read This Manual	It is assumed that the readers of this manual have general knowledge of electricity, logic circuits, and microcontrollers.	
Conventions	Deata significance:	Higher digits on the left and lower digits on the right
	Active low representation:	$\overline{\text{XXX}}$ (overscore over pin or signal name)
	Note:	Footnote for item marked with Note in the text
	Caution:	Information requiring particular attention
	Remark:	Supplementary information
	Numeric representation:	Decimal ... XXXX Hexadecimal ... 0xXXXX

All trademarks and registered trademarks are the property of their respective owners.

TABLE OF CONTENTS

1. GENERAL 5

1.1 Overview 5
1.2 Features 5
1.3 Cautions 5

2. OUTPUT FILES 6

2.1 Description 6

3. INITIALIZATION 19

4. API FUNCTIONS 20

4.1 Overview 20
4.2 Function Reference 21
4.2.1 Common 23
4.2.2 8-Bit timer 33
4.2.3 Buses 41
4.2.4 Clock Frequency Accuracy Measurement Circuit 49
4.2.5 Comparator 58
4.2.6 Compare Match Timer 65
4.2.7 Complementary PWM Mode Timer 75
4.2.8 Continuous Scan Mode S12AD 85
4.2.9 CRC Calculator 100
4.2.10 D/A Converter 110
4.2.11 Data Operation Circuit 119
4.2.12 Data Transfer Controller 129
4.2.13 Dead Time Compensaion Counter 135
4.2.14 DMA Controller 146
4.2.15 Event Link Controller 160
4.2.16 General PWM Timer (GPT) 170
4.2.17 Group Scan Mode S12AD 187
4.2.18 I2C Master Mode 204
4.2.19 I2C Slave Mode 237
4.2.20 Interrupt Controller 253
4.2.21 Low Power Consumption 269
4.2.22 Low Power Timer 281
4.2.23 Normal Mode Timer 287

4.2.24 Phase Counting Mode Timer 295
4.2.25 Port Output Enable 306
4.2.26 I/O Port 317
4.2.27 Programmable Pulse Generator 321
4.2.28 PWM Mode Timer 325
4.2.29 Realtime Clock 333
4.2.30 Remote Control Signal Receiver 360
4.2.31 SCI/SCIF Asynchronous Mode 376
4.2.32 SCI/SCIF Clock Synchronous Mode 400
4.2.33 Single Scan Mode S12AD 423
4.2.34 Smart Card Interface Mode 438
4.2.35 SPI Clock Synchronous Mode 453
4.2.36 SPI Operation Mode 475
4.2.37 Voltage Detection Circuit 490
4.2.38 Watchdog Timer 497
4.2.39 Continuous Scan Mode DSAD 505
4.2.40 Single Scan Mode DSAD 516
4.2.41 Delta-Sigma Modulator Interface 527
4.2.42 Analog Front End 536
4.2.43 Motor 539
4.2.44 LCD Controller 552

Revision Record 559

1. GENERAL

This chapter gives an overview of the driver code generator (hereafter abbreviated as Code Generator) of the Smart Configurator.

1.1 Overview

This tool can output source code (device driver programs as C source and header files) for controlling peripheral modules (clock generation circuit, voltage detection circuit, etc.) of the device by using a GUI to set various types of information on the requirements of the project.

1.2 Features

The features of the Code Generator are as follows.

- Generating code

  The Code Generator outputs not only device driver files in accord with the information set in the GUI but also a complete set of programs for the build environment, such as a sample program containing the call of the main function.

- Reporting

  Information that was set by using the Code Generator can be output to files in various formats and used as design documentation.

- Renaming

  Default names are given to folders and files output by the Code Generator and to the API functions in the source code, but these can be changed to user-specified names.

- Protecting user code

  The user can add user's original source code to each API function. When user generated the device driver programs again by the Code Generator, user's source code within this comment is protected.

  [Comment for user source code descriptions]
  /* Start user code. Do not edit comment generated here */
  /* End user code. Do not edit comment generated here */

  Code written by the user between these comments will be preserved even when the code is generated again.

1.3 Cautions

Smart Configurator has the following cautions.

- OSS (Open Source Software)

  The APIs in this document do not use OSS

2. OUTPUT FILES

This chapter explains the file output by the Code Generator.

2.1 Description

The Code Generator outputs the following files.

Table 2.1 Output File List (1/13)

Peripheral Function	File Name	API Function Name
Common	<workspaceName>.c	main
	dbsct.c	—
	resetprg.c	PowerON_Reset
	sbrk.c	—
	vecttbl.c	—
	vecttbl.h	—
	hwsetup.c	hardware_setup
	hwsetup.h	—
	r_cg_hardware_setup.c	r_undefined_exception R_Systeminit
	r_cg_macrodriver.h	—
	r_cg_userdefine.h	—
	r_smc_entry.h	—
	r_smc_cgc.c	R_CGC_Create
	r_smc_cgc_user.c	R_CGC_Create_UserInit
	r_smc_cgc.h	—
	r_smc_interrup.c	R_Interrupt_Create
	r_smc_interrupt.h	—
	Pin.c	R_Pins_Create
	Pin.h	—

Table 2.2 Output File List (2/13)

Peripheral Function	File Name	API Function Name
8-Bit timer	<Config_TMR0>.c	R_<Config_TMR0>_Create R_<Config_TMR0>_Start R_<Config_TMR0>_Stop
	<Config_TMR0>_user.c	R_<Config_TMR0>_Create_UserInit r_<Config_TMR0>_cmimn_interrupt r_<Config_TMR0>_ovin_interrupt
	<Config_TMR0>.h	–
Buses	<Config_BSC>.c	R_<Config_BSC>_Create R_<Config_BSC>_Error_Monitoring_Start R_<Config_BSC>_Error_Monitoring_Stop R_<Config_BSC>_InitializeSDRAM
	<Config_BSC>_user.c	R_<Config_BSC>_Create_UserInit r_<Config_BSC>_buserr_interrupt
	<Config_BSC>.h	–
Clock Frequency Accuracy Measurement Circuit	<Config_CAC>.c	R_<Config_CAC>_Create R_<Config_CAC>_Start R_<Config_CAC>_Stop
	<Config_CAC>_user.c	R_<Config_CAC>_Create_UserInit r_<Config_CAC>_mendf_interrupt r_<Config_CAC>_mendi_interrupt r_<Config_CAC>_ferrf_interrupt r_<Config_CAC>_ferri_interrupt r_<Config_CAC>_ovff_interrupt r_<Config_CAC>_ovfi_interrupt
	<Config_CAC>.h	–
Comparator	<Config_CMPB0>.c	R_<Config_CMPB0>_Create R_<Config_CMPB0>_Start R_<Config_CMPB0>_Stop
	<Config_CMPB0>user.c	R_<Config_CMPB0>_Create_UserInit r_<Config_CMPB0>_cmpbn_interrupt
	<Config_CMPB0>.h	–
Compare Match Timer	<Config_CMT0>.c	R_<Config_CMT0>_Create R_<Config_CMT0>_Start R_<Config_CMT0>_Stop
	<Config_CMT0>_user.c	R_<Config_CMT0>_Create_UserInit r_<Config_CMT0>_cmin_interrupt r_<Config_CMT0>_cmwin_interrupt r_<Config_CMT0>_icmin_interrupt r_<Config_CMT0>_ocmin_interrupt
	<Config_CMT0>.h	–

Table 2.3 Output File List (3/13)

Peripheral Function	File Name	API Function Name
Complementary PWM Mode Timer	<Config_MTU3_MTU4>.c	R_<Config_MTU3_MTU4>_Create R_<Config_MTU3_MTU4>_Start R_<Config_MTU3_MTU4>_Stop
	<Config_MTU3_MTU4>_user.c	R_<Config_MTU3_MTU4>_Create_UserInit r_<Config_MTU3_MTU4>_tgimn_interrupt r_<Config_MTU3_MTU4>_cj_tgimj_interrupt r_<Config_MTU3_MTU4>_cj_tcivj_interrupt
	<Config_MTU3_MTU4>.h	–
Continuous Scan Mode S12AD	<Config_S12AD0>.c	R_<Config_S12AD0>_Create R_<Config_S12AD0>_Start R_<Config_S12AD0>_Stop R_<Config_S12AD0>_Get_ValueResult R_<Config_S12AD0>_Set_CompareValue R_<Config_S12AD0>_Set_CompareAValue R_<Config_S12AD0>_Set_CompareBValue
	<Config_S12AD0>_user.c	R_<Config_S12AD0>_Create_UserInit r_<Config_S12AD0>_interrupt r_<Config_S12AD0>_compare_interrupt r_<Config_S12AD0>_compare_interruptA r_<Config_S12AD0>_compare_interruptB
	<Config_S12AD0>.h	–
CRC Calculator	<Config_CRC>.c	R_<Config_CRC>_SetCRC8 R_<Config_CRC>_SetCRC16 R_<Config_CRC>_SetCCITT R_<Config_CRC>_SetCRC32 R_<Config_CRC>_SetCRC32C R_<Config_CRC>_Input_Data R_<Config_CRC>_Get_Result
	<Config_CRC.h	–
D/A Converter	<Config_DA>.c	R_<Config_DA>_Create R_<Config_DA>n_Start R_<Config_DA>n_Stop R_<Config_DA>n_Set_ConversionValue R_<Config_DA>_Sync_Start R_<Config_DA>_Sync_Stop
	<Config_DA>_user.c	R_<Config_DA>_Create_UserInit
	<Config_DA>.h	–

Table 2.4 Output File List (4/13)

Peripheral Function	File Name	API Function Name
Data Operation Circuit	<Config_DOC>.c	R_<Config_DOC>_Create R_<Config_DOC>_SetMode R_<Config_DOC>_WriteData R_<Config_DOC>_GetResult R_<Config_DOC>_ClearFlag
	<Config_DOC>_user.c	R_<Config_DOC>_Create_UserInit r_<Config_DOC>_dopcf_interrupt r_<Config_DOC>_dopci_interrupt
	<Config_DOC>.h	–
Data Transfer Controller	<Config_DTC>.c	R_<Config_DTC>_Create R_<Config_DTC>_Start R_<Config_DTC>_Stop
	<Config_DTC>_user.c	R_<Config_DTC>_Create_UserInit
	<Config_DTC>.h	–
Dead Time Compensaion Counter	<Config_MTU5>.c	R_<Config_MTU5>_Create R_<Config_MTU5>_U5_Start R_<Config_MTU5>_U5_Stop R_<Config_MTU5>_V5_Start R_<Config_MTU5>_V5_Stop R_<Config_MTU5>_W5_Start R_<Config_MTU5>_W5_Stop
	<Config_MTU5>_user.c	R_<Config_MTU5>_Create_UserInit r_<Config_MTU5>_tgimn_interrupt
	<Config_MTU5>.h	–
DMA Controller	<Config_DMAC0>.c	R_<Config_DMAC0>_Create R_<Config_DMAC0>_Start R_<Config_DMAC0>_Stop R_<Config_DMAC0>_Set_SoftwareTrigger R_<Config_DMAC0>_Clear_SoftwareTrigger
	<Config_DMAC0>_user.c	R_<Config_DMAC0>_Create_UserInit r_<Config_DMAC0>_dmacni_interrupt r_dmacn_callback_transfer_end r_dmacn_callback_transfer_escape_end
	<Config_DMAC0>.h	–
	r_cg_dmac_user.c	r_dmac_dmac74i_interrupt r_dmacn_callback_transfer_end r_dmacn_callback_transfer_escape_end
	r_cg_dmac.h	–
Event Link Controller	<Config_ELC>.c	R_<Config_ELC>_Create R_<Config_ELC>_Start R_<Config_ELC>_Stop R_<Config_ELC>_GenerateSoftwareEvent R_<Config_ELC>_Set_PortBuffern R_<Config_ELC>_Get_PortBuffern
	<Config_ELC>_user.c	R_<Config_ELC>_Create_UserInit r_<Config_ELC>_elsrni_interrupt
	<Config_ELC>.h	–

Table 2.5 Output File List (5/13)

Peripheral Function	File Name	API Function Name
General PWM Timer	<Config_GPT0>.c	R_<Config_GPT0>_Create R_<Config_GPT0>_Start R_<Config_GPT0>_Stop R_<Config_GPT0>_HardwareStart R_<Config_GPT0>_HardwareStop R_<Config_GPT0>_ETGI_Start R_<Config_GPT0>_ETGI_Stop R_<Config_GPT0>_Software_Clear
	<Config_GPT0>_user.c	R_<Config_GPT0>_Create_UserInit r_<Config_GPT0>_gtcimn_interrupt r_<Config_GPT0>_gtcivn_interrupt r_<Config_GPT0>_gtcinn_interrupt r_<Config_GPT0>_gdten_interrupt
	<Config_GPT0>.h	–
	r_cg_gpt_user.c	r_gpt_etgin_interrupt r_gpt_etgip_interrupt
	r_cg_gpt.h	–
Group Scan Mode S12AD	<Config_S12AD0>.c	R_<Config_S12AD0>_Create R_<Config_S12AD0>_Start R_<Config_S12AD0>_Stop R_<Config_S12AD0>_Get_ValueResult R_<Config_S12AD0>_Set_CompareValue R_<Config_S12AD0>_Set_CompareAValue R_<Config_S12AD0>_Set_CompareBValue
	<Config_S12AD0>_user.c	R_<Config_S12AD0>_Create_UserInit r_<Config_S12AD0>_interrupt r_<Config_S12AD0>_compare_interrupt r_<Config_S12AD0>_compare_interruptA r_<Config_S12AD0>_compare_interruptB r_<Config_S12AD0>_groupb_interrupt r_<Config_S12AD0>_groupc_interrupt
	<Config_S12AD0>.h	–

Table 2.6 Output File List (6/13)

Peripheral Function	File Name	API Function Name
I2C Master Mode	<Config_RIIC0>.c	R_<Config_RIIC0>_Create R_<Config_RIIC0>_Start R_<Config_RIIC0>_Stop R_<Config_RIIC0>_Master_Send R_<Config_RIIC0>_Master_Send_Without_Stop R_<Config_RIIC0>_Master_Receive R_<Config_RIIC0>_IIC_StartCondition R_<Config_RIIC0>_IIC_StopCondition
	<Config_RIIC0>_user.c	R_<Config_RIIC0>_Create_UserInit r_<Config_RIIC0>_error_interrupt r_<Config_RIIC0>_receive_interrupt r_<Config_RIIC0>_transmit_interrupt r_<Config_RIIC0>_transmitend_interrupt r_<Config_RIIC0>_callback_error r_<Config_RIIC0>_callback_transmitend r_<Config_RIIC0>_callback_receiveend
	<Config_RIIC0>.h	–
	<Config_SCI0>.c	R_<Config_SCI0>_Create R_<Config_SCI0>_Start R_<Config_SCI0>_Stop R_<Config_SCI0>_IIC_Master_Send R_<Config_SCI0>_IIC_Master_Receive R_<Config_SCI0>_IIC_StartCondition R_<Config_SCI0>_IIC_StopCondition
	<Config_SCI0>_user.c	R_<Config_SCI0>_Create_UserInit r_<Config_SCI0>_receive_interrupt r_<Config_SCI0>_transmit_interrupt r_<Config_SCI0>_transmitend_interrupt r_<Config_SCI0>_callback_transmitend r_<Config_SCI0>_callback_receiveend
	<Config_SCI0>.h	–
I2C Slave Mode	<Config_RIIC0>.c	R_<Config_RIIC0>_Create R_<Config_RIIC0>_Start R_<Config_RIIC0>_Stop R_<Config_RIIC0>_Slave_Send R_<Config_RIIC0>_Slave_Receive
	<Config_RIIC0>_user.c	R_<Config_RIIC0>_Create_UserInit r_<Config_RIIC0>_error_interrupt r_<Config_RIIC0>_receive_interrupt r_<Config_RIIC0>_transmit_interrupt r_<Config_RIIC0>_transmitend_interrupt r_<Config_RIIC0>_callback_error r_<Config_RIIC0>_callback_transmitend r_<Config_RIIC0>_callback_receiveend
	<Config_RIIC0>.h	–

Table 2.7 Output File List (7/13)

Peripheral Function	File Name	API Function Name
Interrupt Controller	<Config_ICU>.c	R_<Config_ICU>_Create R_<Config_ICU>_IRQn_Start R_<Config_ICU>_IRQn_Stop R_<Config_ICU>_Software_Start R_<Config_ICU>_Software_Stop R_<Config_ICU>_SoftwareInterrupt_Generate R_<Config_ICU>_Software2_Start R_<Config_ICU>_Software2_Stop R_<Config_ICU>_SoftwareInterrupt2_Generate
	<Config_ICU>_user.c	R_<Config_ICU>_Create_UserInit r_<Config_ICU>_irqn_interrupt r_<Config_ICU>_software_interrupt r_<Config_ICU>_software2_interrupt r_<Config_ICU>_nmi_interrupt
	<Config_ICU>.h	–
Low Power Consumption	<Config_LPC>.c	R_<Config_LPC>_Create R_<Config_LPC>_AllModuleClockStop R_<Config_LPC>_Sleep R_<Config_LPC>_DeepSleep R_<Config_LPC>_SoftwareStandby R_<Config_LPC>_DeepSoftwareStandby R_<Config_LPC>_ChangeOperatingPowerControl R_<Config_LPC>_ChangeSleepModeReturnClock R_<Config_LPC>_SetVOLSR_PGAVLS
	<Config_LPC>_user.c	R_<Config_LPC>_Create_UserInit
	<Config_LPC>.h	–
Low Power Timer	<Config_LPT>.c	R_<Config_LPT>_Create R_<Config_LPT>_Start R_<Config_LPT>_Stop
	<Config_LPT>_user.c	R_<Config_LPT>_Create_UserInit
	<Config_LPT>.h	–
Normal Mode Timer	<Config_MTU0>.c	R_<Config_MTU0>_Create R_<Config_MTU0>_Start R_<Config_MTU0>_Stop
	<Config_MTU0>_user.c	R_<Config_MTU0>_Create_UserInit r_<Config_MTU0>_tgimn_interrupt r_<Config_MTU0>_tginm_interrupt r_<Config_MTU0>_tcivn_interrupt r_<Config_MTU0>_tcinv_interrupt
	<Config_MTU0>.h	–

Table 2.8 Output File List (8/13)

Peripheral Function	File Name	API Function Name
Phase Counting Mode Timer	<Config_MTU1>.c	R_<Config_MTU1>_Create R_<Config_MTU1>_Start R_<Config_MTU1>_Stop R_<Config_MTU1_MTU2>_MTU_Start R_<Config_MTU1_MTU2>_MTU_Stop
	<Config_MTU1>_user.c	R_<Config_MTU1>_Create_UserInit r_<Config_MTU1>_tgimn_interrupt r_<Config_MTU1>_tginm_interrupt r_<Config_MTU1>_tcivn_interrupt r_<Config_MTU1>_tcinv_interrupt r_<Config_MTU1>_tciun_interrupt r_<Config_MTU1>_tcinu_interrupt
	<Config_MTU1>.h	–
Port Output Enable	<Config_POE>.c	R_<Config_POE>_Create R_<Config_POE>_Start R_<Config_POE>_Stop R_<Config_POE>_Set_HiZ_MTUn R_<Config_POE>_Clear_HiZ_MTUn R_<Config_POE>_Set_HiZ_GPTn R_<Config_POE>_Clear_HiZ_GPTn
	<Config_POE>_user.c	R_<Config_POE>_Create_UserInit r_<Config_POE>_oein_interrupt
	<Config_POE>.h	–
I/O Port	<Config_PORT>.c	R_<Config_PORT>_Create
	<Config_PORT>_user.c	R_<Config_PORT>_Create_UserInit
	<Config_PORT>.h	–
Programmable Pulse Generator	<Config_PPG0>.c	R_<Config_PPG0>_Create
	<Config_PPG0>_user.c	R_<Config_PPG0>_Create_UserInit
	<Config_PPG0>.h	–
PWM Mode Timer	<Config_MTU0>.c	R_<Config_MTU0>_Create R_<Config_MTU0>_Start R_<Config_MTU0>_Stop
	<Config_MTU0>_user.c	R_<Config_MTU0>_Create_UserInit r_<Config_MTU0>_tgimn_interrupt r_<Config_MTU0>_tginm_interrupt r_<Config_MTU0>_tcivn_interrupt r_<Config_MTU0>_tcinv_interrupt
	<Config_MTU0>.h	–

Table 2.9 Output File List (9/13)

Peripheral Function	File Name	API Function Name
Realtime Clock	<Config_RTC>.c	R_<Config_RTC>_Create R_<Config_RTC>_Start R_<Config_RTC>_Stop R_<Config_RTC>_Restart R_<Config_RTC>_Restart_BinaryCounter R_<Config_RTC>_Set_CalendarCounterValue R_<Config_RTC>_Get_CalendarCounterValue R_<Config_RTC>_Get_CalendarTimeCaptureValuen R_<Config_RTC>_Set_BinaryCounterValue R_<Config_RTC>_Get_BinaryCounterValue R_<Config_RTC>_Get_BinaryTimeCaptureValuen R_<Config_RTC>_Set_RTCOUTOn R_<Config_RTC>_Set_RTCOUTOff R_<Config_RTC>_Set_CalendarAlarm R_<Config_RTC>_Set_BinaryAlarm R_<Config_RTC>_Set_ConstPeriodInterruptOn R_<Config_RTC>_Set_ConstPeriodInterruptOff R_<Config_RTC>_Set_CarryInterruptOn R_<Config_RTC>_Set_CarryInterruptOff R_<Config_RTC>_Enable_Alarm_Interrupt R_<Config_RTC>_Disable_Alarm_Interrupt
	<Config_RTC>_user.c	R_<Config_RTC>_Create_UserInit r_<Config_RTC>_alm_interrupt r_<Config_RTC>_prd_interrupt r_<Config_RTC>_cup_interrupt
	<Config_RTC>.h	–
Remote Control Signal Receiver	<Config_REMC0>.c	R_<Config_REMC0>_Create R_<Config_REMC0>_Start R_<Config_REMC0>_Stop R_<Config_REMC0>_Read
	<Config_REMC0>_user.c	R_<Config_REMC0>_Create_UserInit r_<Config_REMC0>_remcin_interrupt r_<Config_REMC0>_callback_comparematch r_<Config_REMC0>_callback_receiveerror r_<Config_REMC0>_callback_receiveend r_<Config_REMC0>_callback_bufferfull r_<Config_REMC0>_callback_header r_<Config_REMC0>_callback_data0 r_<Config_REMC0>_callback_data1 r_<Config_REMC0>_callback_specialdata
	<Config_REMC0>.h	–

Table 2.10 Output File List (10/13)

Peripheral Function	File Name	API Function Name
SCI/SCIF Asynchronous Mode	<Config_SCI0>.c	R_<Config_SCI0>_Create R_<Config_SCI0>_Start R_<Config_SCI0>_Stop R_<Config_SCI0>_Serial_Send R_<Config_SCI0>_Serial_Receive R_<Config_SCI0>_Serial_Multiprocessor_Send R_<Config_SCI0>_Serial_Multiprocessor_Receive
	<Config_SCI0>_user.c	R_<Config_SCI0>_Create_UserInit r_<Config_SCI0>_transmitend_interrupt r_<Config_SCI0>_transmit_interrupt r_<Config_SCI0>_receive_interrupt r_<Config_SCI0>_receiveerror_interrupt r_<Config_SCI0>_teif_interrupt r_<Config_SCI0>_txif_interrupt r_<Config_SCI0>_rxif_interrupt r_<Config_SCI0>_drif_interrupt r_<Config_SCI0>_erif_interrupt r_<Config_SCI0>_brif_interrupt r_<Config_SCI0>_callback_transmitend r_<Config_SCI0>_callback_receiveend r_<Config_SCI0>_callback_receiveerror r_<Config_SCI0>_callback_error
	<Config_SCI0>.h	–
SCI/SCIF Clock Synchronous Mode	<Config_SCI0>.c	R_<Config_SCI0>_Create R_<Config_SCI0>_Start R_<Config_SCI0>_Stop R_<Config_SCI0>_Serial_Send R_<Config_SCI0>_Serial_Receive R_<Config_SCI0>_Serial_Send_Receive
	<Config_SCI0>_user.c	R_<Config_SCI0>_Create_UserInit r_<Config_SCI0>_transmitend_interrupt r_<Config_SCI0>_transmit_interrupt r_<Config_SCI0>_receive_interrupt r_<Config_SCI0>_receiveerror_interrupt r_<Config_SCI0>_teif_interrupt r_<Config_SCI0>_txif_interrupt r_<Config_SCI0>_rxif_interrupt r_<Config_SCI0>_erif_interrupt r_<Config_SCI0>_brif_interrupt r_<Config_SCI0>_callback_transmitend r_<Config_SCI0>_callback_receiveend r_<Config_SCI0>_callback_receiveerror r_<Config_SCI0>_callback_error
	<Config_SCI0>.h	–

Table 2.11 Output File List (11/13)

Peripheral Function	File Name	API Function Name
Single Scan Mode S12AD	<Config_S12AD0>.c	R_<Config_S12AD0>_Create R_<Config_S12AD0>_Start R_<Config_S12AD0>_Stop R_<Config_S12AD0>_Get_ValueResult R_<Config_S12AD0>_Set_CompareValue R_<Config_S12AD0>_Set_CompareAValue R_<Config_S12AD0>_Set_CompareBValue
	<Config_S12AD0>_user.c	R_<Config_S12AD0>_Create_UserInit r_<Config_S12AD0>_interrupt r_<Config_S12AD0>_compare_interrupt r_<Config_S12AD0>_compare_interruptA r_<Config_S12AD0>_compare_interruptB
	<Config_S12AD0>.h	–
Smart Card Interface Mode	<Config_SCI0>.c	R_<Config_SCI0>_Create R_<Config_SCI0>_Start R_<Config_SCI0>_Stop R_<Config_SCI0>_SmartCard_Send R_<Config_SCI0>_SmartCard_Receive
	<Config_SCI0>_user.c	R_<Config_SCI0>_Create_UserInit r_<Config_SCI0>_transmit_interrupt r_<Config_SCI0>_receive_interrupt r_<Config_SCI0>_receiveerror_interrupt r_<Config_SCI0>_callback_transmitend r_<Config_SCI0>_callback_receiveend r_<Config_SCI0>_callback_receiveerror
	<Config_SCI0>.h	–
SPI Clock Synchronous Mode	<Config_RSPI0>.c	R_<Config_RSPI0>_Create R_<Config_RSPI0>_Start R_<Config_RSPI0>_Stop R_<Config_RSPI0>_Send R_<Config_RSPI0>_Send_Receive R_<Config_RSPI0>_SPI_Master_Send R_<Config_RSPI0>_SPI_Master_Send_Receive R_<Config_RSPI0>_SPI_Slave_Send R_<Config_RSPI0>_SPI_Slave_Send_Receive
	<Config_RSPI0>_user.c	R_<Config_RSPI0>_Create_UserInit r_<Config_RSPI0>_receive_interrupt r_<Config_RSPI0>_transmit_interrupt r_<Config_RSPI0>_error_interrupt r_<Config_RSPI0>_idle_interrupt r_<Config_RSPI0>_transmitend_interrupt r_<Config_RSPI0>_receiveerror_interrupt r_<Config_RSPI0>_callback_receiveend r_<Config_RSPI0>_callback_transmitend r_<Config_RSPI0>_callback_error
	<Config_RSPI0>.h	–

Table 2.12 Output File List (12/13)

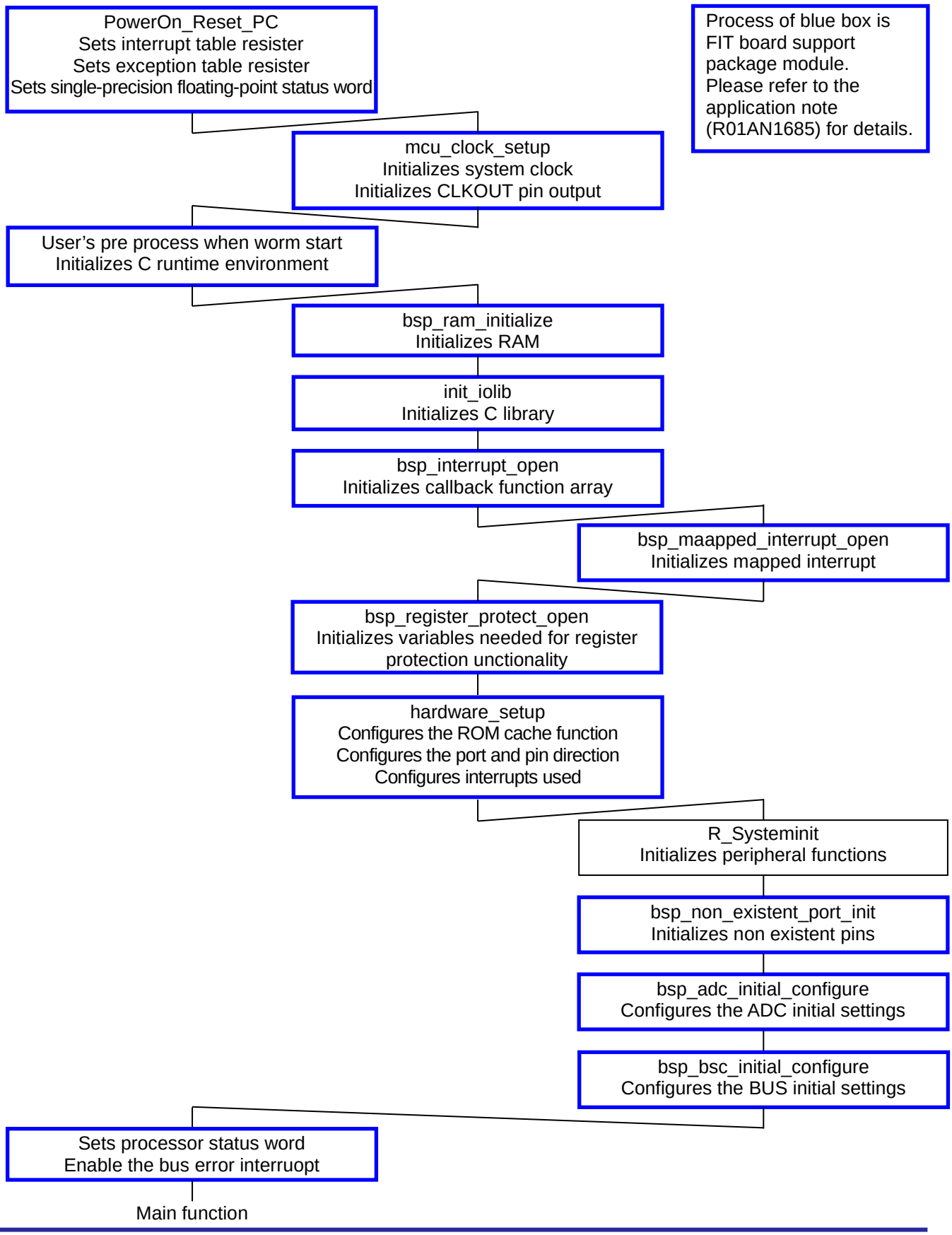
Peripheral Function	File Name	API Function Name
SPI Operation Mode	<Config_RSPI0>.c	R_<Config_RSPI0>_Create R_<Config_RSPI0>_Start R_<Config_RSPI0>_Stop R_<Config_RSPI0>_Send R_<Config_RSPI0>_Send_Receive
	<Config_RSPI0>_user.c	R_<Config_RSPI0>_Create_UserInit r_<Config_RSPI0>_receive_interrupt r_<Config_RSPI0>_transmit_interrupt r_<Config_RSPI0>_error_interrupt r_<Config_RSPI0>_idle_interrupt r_<Config_RSPI0>_callback_receiveend r_<Config_RSPI0>_callback_transmitend r_<Config_RSPI0>_callback_error
	<Config_RSPI0>.h	–
Voltage Detection Circuit	<Config_LVD1>.c	R_<Config_LVD1>_Create R_<Config_LVD1>_Start R_<Config_LVD1>_Stop
	<Config_LVD1>_user.c	R_<Config_LVD1>_Create_UserInit r_<Config_LVD1>_lvdn_interrupt
	<Config_LVD1>.h	–
Watchdog Timer	<Config_WDT>.c	R_<Config_WDT>_Create R_<Config_WDT>_Restart
	<Config_WDT>_user.c	R_<Config_WDT>_Create_UserInit r_<Config_WDT>_wuni_interrupt r_<Config_WDT>_iwuni_interrupt r_<Config_WDT>_nmi_interrupt
	<Config_WDT>.h	–
Continuous Scan Mode DSAD	<Config_DSAD0>.c	R_<Ccnfig_DSAD0>_Create R_<Config_DSAD0>_Start R_<Config_DSAD0>_Stop R_<Config_DSAD0>_Set_SoftwareTrigger R_<Config_DSAD0>_Get_ValueResult R_<Config_DSAD0>_Chm_Set_DisconnectDetection
	<Config_DSAD0>_user.c	R_<Config_DSAD0>_Create_UserInit r_<Config_DSAD0>_adin_interrupt r_<Config_DSAD0>_scanendn_interrupt
	<Config_DSAD0>.h	-
Single Scan Mode DSAD	<Config_DSAD0>.c	R_<Ccnfig_DSAD0>_Create R_<Config_DSAD0>_Start R_<Config_DSAD0>_Stop R_<Config_DSAD0>_Set_SoftwareTrigger R_<Config_DSAD0>_Get_ValueResult R_<Config_DSAD0>_Chm_Set_DisconnectDetection
	<Config_DSAD0>_user.c	R_<Config_DSAD0>_Create_UserInit r_<Config_DSAD0>_adin_interrupt r_<Config_DSAD0>_scanendn_interrupt
	<Config_DSAD0>.h	-

Table 2.13 Output File List (13/13)

Peripheral Function	File Name	API Function Name
Delta-Sigma Modulator Interface	<Config_DSMIF0>.c	R_<Config_DSMIF0>_Create R_<Config_DSMIF0>_Start R_<Config_DSMIF0>_Stop
	<Config_DSMIF0>_user.c	R_<Config_DSMIF0>_Create_UserInit r_<Config_DSMIF0>_ocdin_interrupt r_<Config_DSMIF0>_scdin_interrupt r_<Config_DSMIF0>_sumein_interrupt
	<Config_DSMIF0>.h	–
Analog Front End	<Config_AFE>.c	R_<Config_AFE>_Create
	<Config>AFE>_user.c	R_<Config_AFE>_Create_UserInit
	<Config_AFE>.h	–
Motor (Complementary PWM Mode Timer, Single Scan Mode S12AD)	<Config_MTU3_MTU4>.c	R_<Config_MTU3_MTU4>_Create R_<Config_MTU3_MTU4>_StartTimerCount R_<Config_MTU3_MTU4>_StopTimerCount R_<Config_MTU3_MTU4>_StartTimerCtrl R_<Config_MTU3_MTU4>_StopTimerCtrl R_<Config_MTU3_MTU4>_UpdDuty R_<Config_MTU3_MTU4>_StartAD R_<Config_MTU3_MTU4>_StopAD R_<Config_MTU3_MTU4>_AdcGetConvVal
	<Config_MTU3_MTU4>_user.c	R_<Config_MTU3_MTU4>_Create_UserInit r_<Config_MTU3_MTU4>_CrestInterrupt r_<Config_MTU3_MTU4>_ad_interrupt
	<Config_MTU3_MTU4>.h	–
LCD Controller	<Config_LCD>.c	R_<Config_LCD>_Create R_<Config_LCD>_Start R_<Config_LCD>_Stop R_<Config_LCD>_Voltage_On R_<Config_LCD>_Voltage_Off
	<Config_LCD>_user.c	R_<Config_LCD>_Create_UserInit
	<Config_LCD>.h	-

3. INITIALIZATION

This chapter describes the flow of initialization by the API functions of the Code Generator.



4. API FUNCTIONS

This chapter describes the API functions output that are output by the Code Generator.

4.1 Overview

The following are the naming conventions for the API functions output by the Code Generator.

- Macro names
  These are in all-capital letters.
  Note that if a name includes a number as a prefix, the relevant number is equal to the hexadecimal value of the macro.

- Local variable names
  These are in low-case letters only.

- Global variable names
  These are prefixed with “g”, and only the first letters of words that are elements of the names are capitals.

- Names of pointers to global variables
  These are prefixed with “gp”, and only the first letters of words that are elements of the names are capitals.

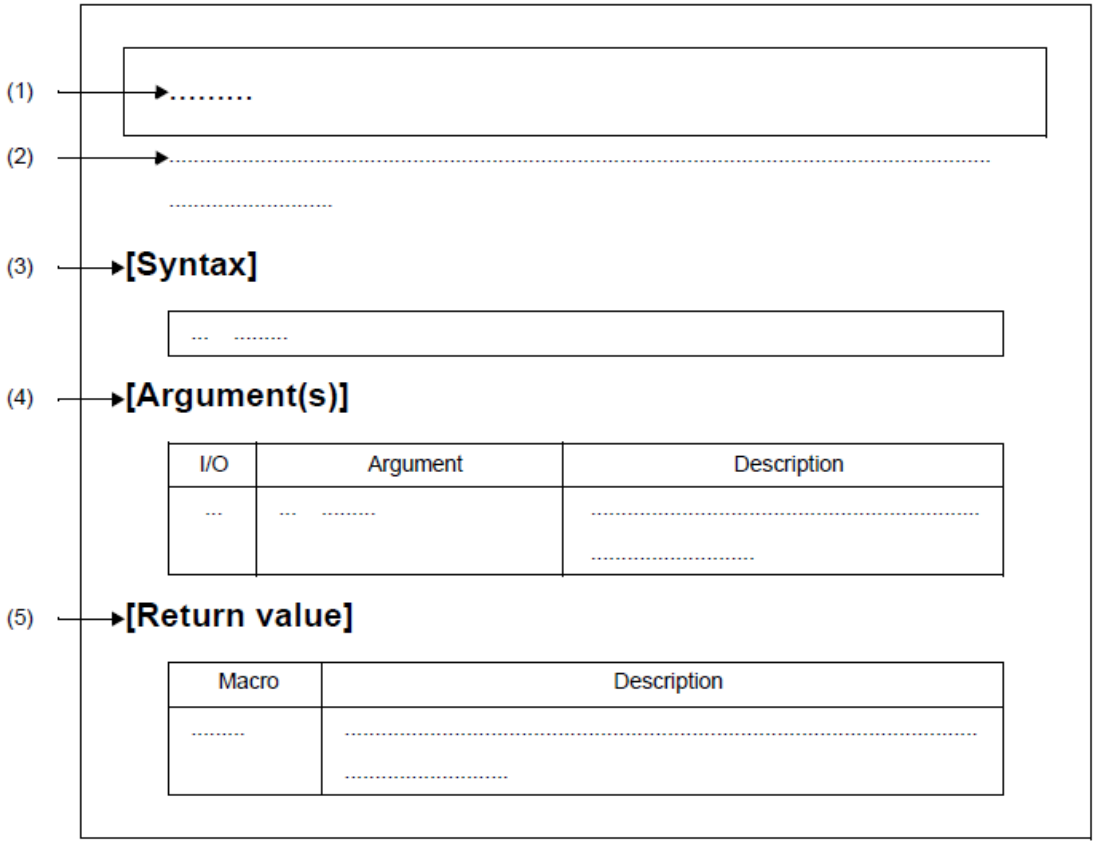
- Names of elements in enumeration specifiers “enum”
  These are in all-capital letters.

Remarks In the generated code by the Smart Configurator, the for statement, the while statement, the do-while statement (loop processing) are used in register setting reflected waiting process etc. If fail-safe processing for infinite loop is required, check the generated code and add processing. Search for key word “WAIT_LOOP” to identify the potential code with infinite loop.

4.2 Function Reference

This section describes the API functions output by the Code Generator, using the following notation format.

Figure 4.1 Notation Format of API Functions



(1) Name
Indicates the name of the API function.

(2) Outline
Outlines the functions of the API function

(3) [Syntax]
Indicates the format to be used when describing an API function to be called in C language.

(4) [Argument(s)]
API function arguments are explained in the following format.

I/O	Argument	Descripton
(a)	(b)	(c)

(a) I/O
Argument classification
I … Input argument
O … Output argument

(b) Argument
Argument data type

(c) Description
Description of argument

(5) [Return value]

API function return value is explained in the following format.

Macro	Description
(a)	(b)

(a) Macro

Macro of return value

(b) Description

Description of return value

4.2.1 Common

The Code Generator outputs the following API functions that are for common use by all peripheral modules.

Table 4.1 Common API Functions

API Function Name	Description
r_undefined_exception	Executes processing in response to undefined instruction exceptions.
PowerON_Reset	Executes processing in response to a reset having been applied.
hardware_setup	Executes initialization processing that is required before controlling various hardware facilities.
R_Systeminit	Executes initialization processing that is required before controlling various peripheral modules.
R_CGC_Create	Executes initialization processing that is required before controlling the clock generation circuit.
R_CGC_Create_UserInit	Executes user-specific initialization processing for the clock generation circuit.
R_Interrupt_Create	Applies the settings for group interrupts and fast interrupts that were specified on the [Interrupts] tabbed page.
R_Pins_Create	Applies the settings for the multi-function pin controller that were specified on the [Pins] tabbed page.
main	main function.

r_undefined_exception

This API function executes processing in response to undefined instruction exceptions.

Remark: This API function is called as the interrupt handler for undefined instruction exceptions, which occur when the attempted execution of an undefined instruction (an instruction that is not implemented) is detected.

[Syntax]

```
void    r_undefined_exception ( void );
```

[Argument(s)]

None.

[Return value]

None.

PowerON_Reset

This API function executes processing in response to a reset having been applied.

Remark: This API function is called as the interrupt handler for internal resets generated by the power-on reset circuit.

[Syntax]

```
void    PowerON_Reset ( void );
```

[Argument(s)]

None.

[Return value]

None.

hardware_setup

This API function executes initialization processing that is required before controlling various hardware facilities.

Remark: This API function is called from PowerON_Reset as a callback routine.

[Syntax]

```
void    hardware_setup ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_Systeminit

This API function executes initialization processing that is required before controlling various peripheral modules.

Remark: This API function is called from hardware_setup as a callback routine.

[Syntax]

```
void    R_Systeminit ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_CGC_Create

This API function executes initialization processing that is required before controlling the clock generation circuit.

Remark: This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_CGC_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_CGC_Create_UserInit

This API function executes user-specific initialization processing for the clock generation circuit.

Remark: This API function is called from R_CGC_Create as a callback routine.

[Syntax]

```
void    R_CGC_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_Interrupt_Create

This API function makes settings for group interrupts and fast interrupts.

Remark: This API function is called fromR_Systeminit before the main() function is executed.

[Syntax]

```
void    R_Interrupt_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_Pins_Create

This API function makes settings for the multi-function pin controller.

[Syntax]

```
void    R_Pin_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

main

This is the main function.

Remark This API function is called from PowerON_Reset as a callback routine.

[Syntax]

```
void    main ( void );
```

[Argument(s)]

None.

[Return value]

None.

4.2.2 8-Bit timer

The Code Generator outputs the following API functions for the 8-bit timer.

Table 4.2 API Functions: [8-bit timer]

API Function Name	Function
R_<Config_TMR0>_Create	Executes initialization processing that is required before controlling the 8-bit timer.
R_<Config_TMR0>_Start	Starts counting by the counter.
R_<Config_TMR0>_Stop	Stops counting by the counter.
R_<Config_TMR0>_Create_UserInit	Executes user-specific initialization processing for the 8-bit timer.
r_<Config_TMR0>_cmimn_interrupt	Executes processing in response to compare match interrupts.
r_<Config_TMR0>_ovin_interrupt	Executes processing in response to overflow interrupts.

R_<Config_TMR0>_Create

This API function executes initialization processing that is required before controlling the 8-bit timer.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_<Config_TMR0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_TMR0>_Start

This API function Starts counting by the counter.

[Syntax]

```
void    R_<Config_TMR0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_TMR0>_Stop

This API function stops counting by the counter.

[Syntax]

```
void    R_<Config_TMR0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_TMR0>_Create_UserInit

This API function executes user-specific initialization processing for the 8-bit timer.

Remark This API function is called from R_<Config_TMR0>_Create as a callback routine.

[Syntax]

```
void R_<Config_TMR0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_TMR0>_cmi*mn*_interrupt

The API function executes processing in response to compare match interrupts.

[Syntax]

```
static void     r_<Config_TMR0>_cmimn_interrupt ( void );
```

Remark *n* is the channel number and *m* is the number of a timer constant register.

[Argument(s)]

None.

[Return value]

None.

r_<Config_TMR0>_ovi*n*_interrupt

This API function executes processing in response to overflow interrupts.

[Syntax]

```
static void      r_<Config_TMR0>_ovin_interrupt ( void );
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

Use timer as One-shot timer.

[API setting example]

r_cg_main.c

```
void main(void)
{
    R_MAIN_UserInit();
    /* Start user code. Do not edit comment generated here */
    /* Start TMR channel 0 counter */
    R_TMR0_Start();

    while (1U)
    {
        ;
    }
    /* End user code. Do not edit comment generated here */
}
```

r_cg_tmr_user.c

```
static void r_tmr_cmia0_interrupt(void)
{
    /* Start user code. Do not edit comment generated here */
    /* Stop TMR channel 0 counter */
    R_TMR0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.3 Buses

The Code Generator outputs the following API functions for the bus.

Table4.3 API Functions: [Buses]

API Function Name	Function
R_<Config_BSC>_Create	Executes initialization processing that is required before controlling the bus.
R_<Config_BSC>_Error_Monitoring_Start	Enables the detection of bus errors due to access to illegal addresses.
R_<Config_BSC>_Error_Monitoring_Stop	Disables the detection of bus errors due to access to illegal addresses.
R_<Config_BSC>_InitializeSDRAM	Initializes the SDRAM controller.
R_<Config_BSC>_Create_UserInit	Executes user-specific initialization processing for the bus.
r_<Config_BSC>_buserr_interrupt	Executes processing in response to bus errors due to access to illegal addresses.

R_<Config_BSC>_Create

This API function executes initialization processing that is required before controlling the bus.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_BSC>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_BSC>_Error_Monitoring_Start

This API function enables the detection of bus errors due to access to illegal addresses.

[Syntax]

```
void    R_<Config_BSC>_Error_Monitoring_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_BSC>_Error_Monitoring_Stop

This API function disables the detection of bus errors due to access to illegal addresses.

[Syntax]

```
void        R_<Config_BSC>_Error_Monitoring_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_BSC>_InitializeSDRAM

his API function initializes the SDRAM controller.

[Syntax]

```
void        R_<Config_BSC>_InitializeSDRAM ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_BSC>_Create_UserInit

This API function executes user-specific initialization processing for the bus.

Remark This API function is called from R_<Config_BSC>_Create as a callback routine.

[Syntax]

```
void R_<Config_BSC>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_BSC>_buserr_interrupt

This API function executes processing in response to bus errors due to access to illegal addresses.

Remark1 This API function is called as the interrupt handler for bus errors due to access by a program to locations in illegal address areas.

Remark2 This API function can be used to determine the bus master that caused the current bus error; to do so, write the processing for reading the MST bits in bus error status register 1 (BERSR1) within this function.

Remark3 This API function can be used to determine the illegal address (the high-order 13 bits of the address) to which access caused the current bus error; to do so, write the processing for reading the ADDR bits in bus error status register 2 (BERSR2) within this function.

[Syntax]

```
void        r_<Config_BSC>_buserr_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Acquiring the address to which access caused a bus error:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Enable BUSERR interrupt in ICU */
    R_Config_BSC_Error_Monitoring_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_BSC_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t g_bsc_buserr_addr;
/* End user code. Do not edit comment generated here */

void r_Config_BSC_buserr_interrupt(void)
{
    /* Start user code for r_Config_BSC_buserr_interrupt. Do not edit comment generated here */
    /* Restore an address that was accessed when a bus error occurred */
    if (1U == BSC.BERSR1.BIT.IA)
    {
        g_bsc_buserr_addr = ((uint16_t)(BSC.BERSR2.WORD)>>3U);
    }

    /* Clear the bus error status registers */
    BSC.BERCLR.BIT.STSCLR = 1U;
    /* End user code. Do not edit comment generated here */
}
```

4.2.4 Clock Frequency Accuracy Measurement Circuit

The Code Generator outputs the following API functions for the clock frequency accuracy measurement circuit (CAC).

Table4.4 API Functions: [Clock Frequency Accuracy Measurement Circuit]

API Function Name	Function
R_<Config_CAC>_Create	Executes initialization processing that is required before controlling the clock frequency accuracy measurement circuit.
R_<Config_CAC>_Start	Starts measurement of the clock frequency accuracy.
R_<Config_CAC>_Stop	Stops measurement of the clock frequency accuracy.
R_<Config_CAC>_Create_UserInit	Executes user-specific initialization processing for the clock frequency accuracy measurement circuit.
r_<Config_CAC>_mendf_interrupt	Executes processing in response to measurement end interrupts. (The name of this API function varies with the device group.)
r_<Config_CAC>_mendi_interrupt	
r_<Config_CAC>_ferrf_interrupt	Executes processing in response to frequency error interrupts. (The name of this API function varies with the device group.)
r_<Config_CAC>_ferri_interrupt	
r_<Config_CAC>_ovff_interrupt	Executes processing in response to overflow interrupts. (The name of this API function varies with the device group.)
r_<Config_CAC>_ovfi_interrupt	

R_<Config_CAC>_Create

This API function executes initialization processing that is required before controlling the clock frequency accuracy measurement circuit.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_CAC>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CAC>_Start

This API function starts measurement of the clock frequency accuracy.

[Syntax]

```
void        R_<Config_CAC>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CAC>_Stop

This API function stops measurement of the clock frequency accuracy.

[Syntax]

```
void        R_<Config_CAC>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CAC>_Create_UserInit

This API function executes user-specific initialization processing for the clock frequency accuracy measurement circuit (CAC).

Remark This API function is called from R_<Config_CAC>_Create as a callback routine.

[Syntax]

```
void R_<Config_CAC>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_CAC>_mendf_interrupt

r_<Config_CAC>_mendi_interrupt

This API function executes processing in response to measurement end interrupts.

Remark This API function is called as the interrupt handler for measurement end interrupts, which occur when the clock frequency accuracy measurement circuit detects valid edges on the reference signal line.

[Syntax]

```
void    r_<Config_CAC>_mendf_interrupt ( void );
```

```
void    r_<Config_CAC>_mendi_interrupt ( void );
```

Remark The name of this API function varies with the device group.

[Argument(s)]

None.

[Return value]

None.

r_<Config_CAC>_ferrf_interrupt

r_<Config_CAC>_ferri_interrupt

This API function executes processing in response to measurement end interrupts.

Remark This API function is called as the interrupt handler for measurement end interrupts, which occur when the clock frequency accuracy measurement circuit detects valid edges on the reference signal line.

[Syntax]

```
void    r_<Config_CAC>_ferrf_interrupt ( void );
```

```
void    r_<Config_CAC>_ferri_interrupt ( void );
```

Remark The name of this API function varies with the device group.

[Argument(s)]

None.

[Return value]

None.

r_<Config_CAC>_ovff_interrupt

r_<Config_CAC>_ovfi_interrupt

This API function executes processing in response to overflow interrupts.

Remark This API function is called as the interrupt handler for overflow interrupts, which occur when the counter value overflows.

[Syntax]

```
void    r_<Config_CAC>_ovff_interrupt ( void );
```

```
void    r_<Config_CAC>_ovfi_interrupt ( void );
```

Remark The name of this API function varies with the device group.

[Argument(s)]

None.

[Return value]

None.

Usage example

Counting the number of frequency errors:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Enable clock frequency measurement */
    R_Config_CAC_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_CAC_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_cac_ferri_cnt;
/* End user code. Do not edit comment generated here */

void R_Config_CAC_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    /* Reset the error countor */
    g_cac_ferri_cnt = 0U;
    /* End user code. Do not edit comment generated here */
}

void r_Config_CAC_ferri_interrupt(void)
{
    /* Start user code for r_Config_CAC_ferri_interrupt. Do not edit comment generated here */
    /* Add the error countor */
    g_cac_ferri_cnt++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.5 Comparator

The Code Generator outputs the following API functions for the comparator.

Table4.5 API Functions: [Comparator]

API Function Name	Function
R_<Config_CMPB0>_Create	The Code Generator outputs the following API functions for the comparator.
R_<Config_CMPB0>_Start	The Code Generator outputs the following API functions for the comparator.
R_<Config_CMPB0>_Stop	The Code Generator outputs the following API functions for the comparator.
R_<Config_CMPB0>_Create_UserInit	The Code Generator outputs the following API functions for the comparator.
r_<Config_CMPB0>_cmpbn_interrupt	The Code Generator outputs the following API functions for the comparator.

R_<Config_CMPB0>_Create

This API function executes initialization processing that is required before controlling the comparator.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_CMPB0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CMPB0>_Start

This API function starts comparison of the analog input voltage with the reference voltage.

[Syntax]

```
void        R_<Config_CMPB0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CMPB0>_Stop

This API function stops comparison of the analog input voltage with the reference voltage.

[Syntax]

```
void        R_<Config_CMPB0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CMPB0>_Create_UserInit

This API function executes user-specific initialization processing for the comparator.

Remark This API function is called from R_<Config_CMPB0>_Create as a callback routine.

[Syntax]

```
void R_<Config_CMPB0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_CMPB0>_cmpb*n*_interrupt

This API function executes processing in response to comparator Bn interrupts.

Remark This API function is called as the interrupt handler for comparator Bn interrupts, which occur when the result of comparison of the analog input voltage with the reference input voltage changes.

[Syntax]

```
void r_<Config_CMPB0>_cmpbn_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

Setting a flag when the result of comparison changes:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start comparator B0 */
    R_Config_CMPB0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_CMPB0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_cmpb0_f;
/* End user code. Do not edit comment generated here */

void R_Config_CMPB0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    /* Clear the flag */
    g_cmpb0_f = 0U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_CMPB0_cmpb0_interrupt(void)
{
    /* Start user code for r_Config_CMPB0_cmpb0_interrupt. Do not edit comment generated here
*/
    /* Set the flag */
    g_cmpb0_f = 1U;
    /* End user code. Do not edit comment generated here */
}
```

4.2.6 Compare Match Timer

The Code Generator outputs the following API functions for compare match timers (CMT or CMTW).

Table4.6 API Functions: [Compare Match Timer]

API Function Name	Function
R_<Config_CMT0>_Create	Executes initialization processing that is required before controlling a compare match timer (CMT or CMTW).
R_<Config_CMT0>_Start	Starts counting by the counter.
R_<Config_CMT0>_Stop	Stops counting by the counter.
R_<Config_CMT0>_Create_UserInit	Executes user-specific initialization processing for a compare match timer (CMT or CMTW).
r_<Config_CMT0>_cmin_interrupt	Executes processing in response to compare match interrupts (CMI*n*).
r_<Config_CMT0>_cmwin_interrupt	Executes processing in response to compare match interrupts (CMWI*n*).
r_<Config_CMT0>_icmin_interrupt	Executes processing in response to input capture interrupts.
r_<Config_CMT0>_ocmin_interrupt	Executes processing in response to output compare interrupts.

R_<Config_CMT0>_Create

This API function executes initialization processing that is required before controlling a compare match timer (CMT or CMTW).

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_CMT0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CMT0>_Start

This API function starts counting by the counter.

[Syntax]

```
void        R_<Config_CMT0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CMT0>_Stop

This API function stops counting by the counter.

[Syntax]

```
void        R_<Config_CMT0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CMT0>_Create_UserInit

This API function executes user-specific initialization processing for a compare match timer (CMT or CMTW).

Remark This API function is called from R_<Config_CMT0>_Create as a callback routine.

[Syntax]

```
void R_<Config_CMT0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_CMT0>_cmi*n*_interrupt

This API function executes processing in response to compare match interrupts (CMI*n*).

Remark This API function is called as the interrupt handler for compare match interrupts (CMI*n*), which occur when the current counter value (the value of the compare match timer counter (CMCNT)) matches a specified value (the value of the compare match timer constant register (CMCOR)).

[Syntax]

```
void    r_<Config_CMT0>_cmin_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

r_<Config_CMT0>_cmwi*n*_interrupt

This API function executes processing in response to compare match interrupts (CMWI*n*).

Remark This API function is called as the interrupt handler for compare match interrupts (CMWI*n*), which occur when the current counter value (the value of the compare match timer counter (CMWCNT)) matches a specified value (the value of the compare match timer constant register (CMWCOR)).

[Syntax]

```
void    r_<Config_CMT0>_cmwin_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

r_<Config_CMT0>_icmi*n*_interrupt

This API function executes processing in response to input capture interrupts.

[Syntax]

```
void        r_<Config_CMT0>_icmin_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

r_<Config_CMT0>_ocmi*n*_interrupt

This API function executes processing in response to output compare interrupts.

[Syntax]

```
void        r_<Config_CMT0>_ocmi n_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

With the timer operating in a one-shot manner:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start CMT channel 0 counter */
    R_Config_CMT0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_CMT0_user.c

```
static void r_Config_CMT0_cmi0_interrupt(void)
{
    /* Start user code for r_Config_CMT0_cmi0_interrupt. Do not edit comment generated here */
    /* Stop CMT channel 0 counter */
    R_Config_CMT0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.7 Complementary PWM Mode Timer

The Code Generator outputs the following API functions for timers in complementary PWM mode.

Table4.7 API Functions: [Complementary PWM Mode Timer]

API Function Name	Function
R_<Config_MTU3_MTU4>_Create	Executes initialization processing that is required before controlling a timer in complementary PWM mode.
R_<Config_MTU3_MTU4>_Start	Starts counting by the counter.
R_<Config_MTU3_MTU4>_Stop	Stops counting by the counter.
R_<Config_MTU3_MTU4>_Create_UserInit	Executes user-specific initialization processing for a timer in complementary PWM mode.
r_<Config_MTU3_MTU4>_tgimn_interrupt	Executes processing in response to compare match interrupts.
r_<Config_MTU3_MTU4>_cj_tgimj_interrupt	Executes processing in response to compare match interrupts.
r_<Config_MTU3_MTU4>_cj_tcivj_interrupt	Executes processing in response to underflow interrupts.

R_<Config_MTU3_MTU4>_Create

This API function executes initialization processing that is required before controlling a timer in complementary PWM mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_<Config_MTU3_MTU4>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_Start

This API function starts counting by the counter.

[Syntax]

```
void        R_<Config_MTU3_MTU4>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_Stop

This API function stops counting by the counter.

[Syntax]

```
void        R_<Config_MTU3_MTU4>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_Create_UserInit

This API function executes user-specific initialization processing for a timer in complementary PWM mode.

Remark This API function is called from R_<Config_MTU3_MTU4>_Create as a callback routine.

[Syntax]

```
void R_<Config_MTU3_MTU4>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU3_MTU4>_tgi*mn*_interrupt

This API function executes processing in response to compare match interrupts.

Remark This API function is called as the interrupt handler for compare match interrupts, which occur when the current counter value (the value of the timer counter (TCNT)) matches a specified value (the value of the timer general register (TGR)).

[Syntax]

```
void r_<Config_MTU3_MTU4>_tgimn_interrupt ( void );
```

Remark *n* is channel numbers, and *m* is the number of a timer general register.

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU3_MTU4>_c*j*_tgi*mj*_interrupt

This API function executes processing in response to compare match interrupts.

Remark This API function is called as the interrupt handler for compare match interrupts, which occur when the current counter value (the value of the timer counter (TCNT)) matches a specified value (the value of the timer general register (TGR)).

[Syntax]

```
void    r_<Config_MTU3_MTU4>_cj_tgimj_interrupt ( void );
```

Remark *n* is channel numbers, and *m* is the number of a timer general register.

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU3_MTU4>_c*j*_tciv*j*_interrupt

This API function executes processing in response to compare match interrupts.

Remark This API function is called as the interrupt handler for compare match interrupts, which occur when the current counter value (the value of the timer counter (TCNT)) matches a specified value (the value of the timer general register (TGR)).

[Syntax]

```
void r_<Config_MTU3_MTU4>_cj_tcivj_interrupt ( void );
```

Remark *j* is channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

Repeating processing to gradually increase the width of the U-phase pulses to the upper limit and then gradually reduce it to the lower limit:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the MTU6 channel counter */
    R_Config_MTU6_MTU7_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_MTU6_MTU7_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t gu_pulse_u;
volatile int8_t g_pulse_dir_u;
/* End user code. Do not edit comment generated here */

void R_Config_MTU6_MTU7_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    gu_pulse_u = _xxxx_6TGRB_VALUE;
    g_pulse_dir_u = 1U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_MTU6_MTU7_tgib6_interrupt(void)
{
    /* Start user code for r_Config_MTU6_MTU7_tgib6_interrupt. Do not edit comment generated
here */
    gu_pulse_u += g_pulse_dir_u;
    if(gu_pulse_u == _xxxx_TCDRB_VALUE)
    {
        g_pulse_dir_u = -1;
    }
    else if(gu_pulse_u == _xxxx_TDDRB_VALUE)
    {
        g_pulse_dir_u = 1;
    }
    MTU6.TGRB = gu_pulse_u;
    MTU6.TGRD = gu_pulse_u;
    /* End user code. Do not edit comment generated here */
}
```

4.2.8 Continuous Scan Mode S12AD

The Code Generator outputs the following API functions for the continuous scan mode S12AD.

Table4.8 API Functions: [Countinuous Scan Mode S12AD]

API Function Name	Function
R_<Config_S12AD0>_Create	Executes initialization processing that is required before controlling the S12AD in continuous scan mode.
R_<Config_S12AD0>_Start	Starts A/D conversion.
R_<Config_S12AD0>_Stop	Stops A/D conversion.
R_<Config_S12AD0>_Get_ValueResult	Gets the result of conversion.
R_<Config_S12AD0>_Set_CompareValue	Sets the compare level.
R_<Config_S12AD0>_Set_CompareAValue	Sets the compare level for window A.
R_<Config_S12AD0>_Set_CompareBValue	Sets the compare level for window B.
R_<Config_S12AD0>_Create_UserInit	Executes user-specific initialization processing for the S12AD in continuous scan mode.
r_<Config_S12AD0>_interrupt	Executes processing in response to scan end interrupts.
r_<Config_S12AD0>_compare_interrupt	Executes processing in response to compare interrupts.
r_<Config_S12AD0>_compare_interruptA	Executes processing in response to compare interrupts for window A.
r_<Config_S12AD0>_compare_interruptB	Executes processing in response to compare interrupts for window B.

R_<Config_S12AD0>_Create

This API function executes initialization processing that is required before controlling the continuous scan mode S12AD.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_<Config_S12AD0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Start

This API function starts A/D conversion.

[Syntax]

```
void        R_<Config_S12AD0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Stop

This API function stops A/D conversion.

[Syntax]

```
void        R_<Config_S12AD0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Get_ValueResult

This API function gets the result of conversion.

[Syntax]

```
void    R_<Config_S12AD0>_Get_ValueResult ( ad_channel_t channel, uint16_t * const buffer );
```

[Argument(s)]

For RX130 or RX230/RX231

I/O	Argument	Description
I	ad_channel_t *channel*;	Channel number ADCHANNEL0: Input channel AN000 ADCHANNEL1: Input channel AN001 ADCHANNEL2: Input channel AN002 ADCHANNEL3: Input channel AN003 ADCHANNEL4: Input channel AN004 ADCHANNEL5: Input channel AN005 ADCHANNEL6: Input channel AN006 ADCHANNEL7: Input channel AN007 ADCHANNEL16: Input channel AN016 ADCHANNEL17: Input channel AN017 ADCHANNEL18: Input channel AN018 ADCHANNEL19: Input channel AN019 ADCHANNEL20: Input channel AN020 ADCHANNEL21: Input channel AN021 ADCHANNEL22: Input channel AN022 ADCHANNEL23: Input channel AN023 ADCHANNEL24: Input channel AN024 ADCHANNEL25: Input channel AN025 ADCHANNEL26: Input channel AN026 ADCHANNEL27: Input channel AN027 ADCHANNEL28: Input channel AN028 ADCHANNEL29: Input channel AN029 ADCHANNEL30: Input channel AN030 ADCHANNEL31: Input channel AN031 ADTEMPSENSOR: Extended analog input (temperature sensor output) ADINTERREFVOLT: Extended analog input (internal reference voltage) ADSELFDIAGNOSIS: Result of self-diagnosis ADDATADUPLICATION: Double-trigger mode result
O	uint16_t * const *buffer*;	Pointer to the location where the acquired results of conversion are to be stored

For other devices

I/O	Argument	Description
I	ad_channel_t *channel*;	Channel number ADCHANNEL0: Input channel AN000 ADCHANNEL1: Input channel AN001 ADCHANNEL2: Input channel AN002 ADCHANNEL3: Input channel AN003 ADCHANNEL4: Input channel AN004 ADCHANNEL5: Input channel AN005 ADCHANNEL6: Input channel AN006 ADCHANNEL7: Input channel AN007 ADCHANNEL8: Input channel AN008 ADCHANNEL9: Input channel AN009 ADCHANNEL10: Input channel AN010 ADCHANNEL11: Input channel AN011 ADCHANNEL12: Input channel AN012 ADCHANNEL13: Input channel AN013 ADCHANNEL14: Input channel AN014 ADCHANNEL15: Input channel AN015 ADCHANNEL16: Input channel AN016 ADCHANNEL17: Input channel AN017 ADCHANNEL18: Input channel AN018 ADCHANNEL19: Input channel AN019 ADCHANNEL20: Input channel AN020 ADTEMPSENSOR: Extended analog input (temperature sensor output) ADINTERREFVOLT: Extended analog input (internal reference voltage) ADSELFDIAGNOSIS: Result of self-diagnosis ADDATADUPLICATION: Double-trigger mode result ADDATADUPLICATIONA: Double-trigger mode A result ADDATADUPLICATIONB: Double-trigger mode B result
O	uint16_t * const *buffer*;	Pointer to the location where the acquired results of conversion are to be stored

[Return value]

None.

R_<Config_S12AD0>_Set_CompareValue

This API function sets the compare level.

[Syntax]

```
void    R_<Config_S12AD0>_Set_CompareValue ( uint16_t reg_value0, uint16_t reg_value1);
```

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Set_CompareAValue

This API function sets the compare level for window A.

[Syntax]

void R_<Config_S12AD0>_Set_CompareAValue (uint16_t *reg_value0,* uint16_t *reg_value1*);

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Set_CompareBValue

This API function sets the compare level for window B.

[Syntax]

void R_<Config_S12AD0>_Set_CompareBValue (uint16_t *reg_value0*, uint16_t *reg_value1*);

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Create_UserInit

This API function executes user-specific initialization processing for the continuous scan mode S12AD.

Remark This API function is called from R_<Config_S12AD0>_Create as a callback routine.

[Syntax]

```
void R_<Config_S12AD0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_interrupt

This API function executes processing in response to scan end interrupts.

[Syntax]

```
void        r_<Config_S12AD0>_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interrupt

This API function executes processing in response to compare interrupts.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interruptA

This API function executes processing in response to compare interrupts for window A.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interruptA ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interruptB

This API function executes processing in response to compare interrupts for window B.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interruptB ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Getting the result of A/D conversion that matches the default setting of the condition for comparison and then changing the compare match values:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the AD0 converter */
    R_Config_S12AD0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_S12AD0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t g_s12ad0_ch000_value;
/* End user code. Do not edit comment generated here */

void r_Config_S12AD0_compare_interrupt(void)
{
    /* Start user code for r_Config_S12AD0_compare_interrupt. Do not edit comment generated here */
    /* Stop the AD0 converter */
    R_Config_S12AD0_Stop();

    /* Get result from the AD0 channel 0 (AN000) converter */
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, (uint16_t *)&g_s12ad0_ch000_value);

    /* Set reference data for AD0 comparison */
    R_Config_S12AD0_Set_CompareValue(1000U, 3000U);

    /* Clear the compare flag */
    S12AD.ADCMPSR0.WORD = 0x00U;

    /* Start the AD0 converter */
    R_Config_S12AD0_Start();
    /* End user code. Do not edit comment generated here */
}
```

4.2.9 CRC Calculator

The Code Generator outputs the following API functions for the CRC calculator.

Table4.9 API Functions: [CRC Calculato]

API Function Name	Function
R_<Config_CRC>_SetCRC8	Initializes the CRC calculator in preparation for 8-bit CRC calculation (polynomial: $X^8 + X^2 + X + 1$).
R_<Config_CRC>_SetCRC16	Initializes the CRC calculator in preparation for 16-bit CRC calculation (polynomial: $X^{16} + X^{15} + X^2 + 1$).
R_<Config_CRC>_SetCCITT	Initializes the CRC calculator in preparation for 16-bit CRC calculation (polynomial: $X^{16} + X^{12} + X^5 + 1$).
R_<Config_CRC>_SetCRC32	Initializes the CRC calculator in preparation for 32-bit CRC calculation (polynomial: $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$).
R_<Config_CRC>_SetCRC32C	Initializes the CRC calculator in preparation for 32-bit CRC calculation (polynomial: $X^{32} + X^{28} + X^{27} + X^{26} + X^{25} + X^{23} + X^{22} + X^{20} + X^{19} + X^{18} + X^{14} + X^{13} + X^{11} + X^{10} + X^9 + X^8 + X^6 + 1$).
R_<Config_CRC>_Input_Data	Sets the initial value of the data from which the CRC is to be calculated.
R_<Config_CRC>_Get_Result	Gets the result of the operation.

R_<Config_CRC>_SetCRC8

This API function initializes the CRC calculator in preparation for 8-bit CRC calculation (polynomial: $X^8 + X^2 + X + 1$).

[Syntax]

```
void        R_<Config_CRC>_SetCRC8 ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CRC>_SetCRC16

This API function initializes the CRC calculator in preparation for 16-bit CRC calculation (polynomial: $X^{16} + X^{15} + X^2 + 1$).

[Syntax]

```
void        R_<Config_CRC>_SetCRC16 ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CRC>_SetCCITT

This API function initializes the CRC calculator in preparation for 16-bit CRC calculation (polynomial: $X^{16} + X^{12} + X^5 + 1$).

[Syntax]

```
void        R_<Config_CRC>_SetCCITT ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CRC>_SetCRC32

This API function initializes the CRC calculator in preparation for 32-bit CRC calculation (polynomial: $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$).

[Syntax]

```
void        R_<Config_CRC>_SetCRC32 ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CRC>_SetCRC32C

This API function initializes the CRC calculator in preparation for 32-bit CRC calculation (polynomial: $X^{32} + X^{28} + X^{27} + X^{26} + X^{25} + X^{23} + X^{22} + X^{20} + X^{19} + X^{18} + X^{14} + X^{13} + X^{11} + X^{10} + X^{9} + X^{8} + X^{6} + 1$).

[Syntax]

```
void        R_<Config_CRC>_SetCRC32C ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_CRC>_Input_Data

This API function sets the initial value of the data from which the CRC is to be calculated.

[Syntax]

```
void    R_<Config_CRC>_Input_Data ( uint8_t data );
```

```
void    R_<Config_CRC>_Input_Data ( uint32_t data );
```

[Argument(s)]

I/O	Argument	Description
I	uint8_t *data*;	Initial value of the data from which the CRC is to be calculated

I/O	Argument	Description
I	uint32_t *data*;	Initial value of the data from which the CRC is to be calculated

Remark The sizes of the arguments vary with the device group

[Return value]

None.

R_<Config_CRC>_Get_Result

This API function gets the result of the operation.

[Syntax]

```
void    R_<Config_CRC>_Get_Result ( uint16_t * const result );
```

```
void    R_<Config_CRC>_Get_Result ( uint32_t * const result );
```

[Argument(s)]

I/O	Argument	Description
O	uint16_t * const *result*;	Pointer to the location where the result of the operation is to be stored

I/O	Argument	Description
O	uint32_t * const *result*;	Pointer to the location where the result of the operation is to be stored

Remark The sizes of the arguments vary with the device group.

[Return value]

None.

Usage example1

Generating the CRC code and appending it to data for transmission:

tx_func.c

```
#include "r_smc_entry.h"
volatile uint8_t tx_buf[2];
volatile uint16_t result;
void tx_func(void)
{
    /* Set CRC module using CRC8 algorithm */
    R_Config_CRC_SetCRC8();

    /* Restore transmit data */
    tx_buf[0] = 0xF0;

    /* Write data to CRC input register */
    R_Config_CRC_Input_Data(tx_buf[0]);

    /* Get result from CRC output register */
    R_Config_CRC_Get_Result((uint16_t *)&result);

    /* Restore CRC code */
    tx_buf[1] = (uint8_t)(result);

    /*** Transmit "tx_buf" ***/
}
```

Usage example2

Generating the CRC code from the received data and checking the received data for correctness:

rx_func.c

```
#include "r_smc_entry.h"
volatile uint8_t rx_buf[2];
volatile uint16_t result;
volatile uint8_t err_f;
void rx_func(void)
{
    /* Clare error flag */
    err_f = 0U;

    /*** Receive (Restore the receive data in "rx_buf") ***/

    /* Set CRC module using CRC8 algorithm */
    R_Config_CRC_SetCRC8();

    /* Write data to CRC input register */
    R_Config_CRC_Input_Data(rx_buf[0]);

    /* Get result from CRC output register */
    R_Config_CRC_Get_Result((uint16_t *)&result);

    /* Check the receive data */
    if (rx_buf[1] != (uint8_t)(result))
    {
        /* Set error flag */
        err_f = 1U;
    }
}
```

4.2.10 D/A Converter

The Code Generator outputs the following API functions for the D/A converter.

Table4.10 API Functions: [D/A Convertor]

API Function Name	Function
R_<Config_DA>_Create	Executes initialization processing that is required before controlling the D/A converter.
R_<Config_DA>n_Start	Starts D/A conversion.
R_<Config_DA>n_Stop	Stops D/A conversion.
R_<Config_DA>n_Set_ConversionValue	Sets a value for D/A conversion.
R_<Config_DA>_Sync_Start	Starts synchronous D/A conversion.
R_<Config_DA>_Sync_Stop	Stops synchronous D/A conversion.
R_<Config_DA>_Create_UserInit	Executes user-specific initialization processing for the D/A converter.

R_<Config_DA>_Create

This API function executes initialization processing that is required before controlling the D/A converter.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_DA>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DA>*n*_Start

This API function starts D/A conversion.

Remark When using RX23E-B / RX26T / RX64M / RX651 / RX65N / RX660 / RX66N / RX66T / RX71M / RX72M / RX72N / RX72T, this API function should be called before R_<Config_DA>n_Set_ConversionValue.

[Syntax]

```
void R_<Config_DA>n_Start ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

R_<Config_DA>*n*_Stop

This API function stops D/A conversion.

[Syntax]

```
void        R_<Config_DA>n_Stop ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

R_<Config_DA>*n*_Set_ConversionValue

This API function sets a value for D/A conversion.

[Syntax]

void	R_<Config_DA>*n*_Set_ConversionValue (uint16_t *reg_value*);

Remark *n* is a channel number.

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value*;	Value to be D/A converted

[Return value]

None.

R_<Config_DA>_Sync_Start

This API function starts synchronous D/A conversion.

Remark When using with RX64M / RX651 / RX65N / RX66N / RX71M / RX72M / RX72N, this API function should be called before R_<Config_DA>n_Set_ConversionValue.

[Syntax]

```
void    R_<Config_DA>_Sync_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DA>_Sync_Stop

This API function stops synchronous D/A conversion.

[Syntax]

```
void        R_<Config_DA>_Sync_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DA>_Create_UserInit

This API function executes user-specific initialization processing for the D/A converter.

Remark This API function is called from R_<Config_DA>_Create as a callback routine.

[Syntax]

```
void R_<Config_DA>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Enabling synchronous D/A conversion in channels 0 and 1:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Set the DA0 converter value */
    R_<Config_DA>0_Set_ConversionValue(1000U);

    /* Set the DA1 converter value */
    R_<Config_DA>1_Set_ConversionValue(2000U);

    /* Enable the DA0, DA1 synchronize converter */
    R_<Config_DA>_Sync_Start();

    while (1U)
    {
        nop();
    }
}
```

4.2.11 Data Operation Circuit

The Code Generator outputs the following API functions for the data operation circuit.

Table4.11 API Functions: [Data Operation Circuit]

API Function Name	Function
R_<Config_DOC>_Create	Executes initialization processing that is required before controlling the data operation circuit.
R_<Config_DOC>_SetMode	Sets the operating mode and the initial reference value for use by the data operation circuit.
R_<Config_DOC>_WriteData	Sets the input value (value for comparison with, addition to, or subtraction from the reference value) for use in the operation.
R_<Config_DOC>_GetResult	Gets the result of the operation.
R_<Config_DOC>_ClearFlag	Clears the data operation circuit flag.
R_<Config_DOC>_Create_UserInit	Executes user-specific initialization processing for the data operation circuit.
r_<Config_DOC>_dopcf_interrupt	Executes processing in response to data operation circuit interrupts. (The name of this API function varies with the device group.)
r_<Config_DOC>_dopci_interrupt	

R_<Config_DOC>_Create

This API function executes initialization processing that is required before controlling the data operation circuit.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_<Config_DOC>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DOC>_SetMode

This API function sets the operating mode and the initial reference values for use by the data operation circuit. And there are two types of parameters for this API based on the specific device DOC functions, one is defined with 2 parameters, the other is defined with three parameters.

- R_<Config_DOC>_SetMode with two parameters: “mode” and “value”

Remark1. When COMPARE_MISMATCH (data comparison mode) or COMPARE_MATCH (data comparison mode) is specified as the operating mode *mode*, the 16-bit data *value* is stored in the DOC data setting register (DODSR).

Remark2. When ADDITION (data addition mode) or SUBTRACTION (data subtraction mode) is specified as the operating mode *mode*, the 16-bit data *value* is stored in the DOC data setting register (DODSR) as the initial value.

[Syntax]

void R_<Config_DOC>_SetMode (doc_mode_t *mode*, uint16_t *value*);

[Argument(s)]

I/O	Argument	Description
I	doc_mode_t *mode*;	Operating mode type (including the condition for detection) COMPARE_MISMATCH: Data comparison mode (mismatch) COMPARE_MATCH: Data comparison mode (match) ADDITION: Data addition mode SUBTRACTION: Data subtraction mode
I	uint16_t *value*;	Reference value for comparison operation, and operation result for addition or subtraction

[Return value]

None.

- R_<Config_DOC>_SetMode with three parameters: “mode”, “value1” and “value2”

Remark1. When data comparison mode is specified as the operation mode *mode* and its value is COMPARE_NEQ (not equal) or COMPARE_EQ (equal) or COMPARE_GT (greater than) or COMPARE_LT (less than), the 16-bit/32-bit data value is stored in the DOC data setting register 0 (DODSR0).

Remark2. When ADDITION (data addition mode) or SUBTRACTION (data subtraction mode) is specified as the operating mode *mode*, the 16-bit/32-bit data value is stored in the DOC data setting register 0 (DODSR0) as the initial value

Remark3. When COMPARE_IN_RANGE (data compare mode) or COMPARE_OUT_RANGE (data compare mode) is specified as the operating mode *mode*, the 16-bit/32-bit data value 1 (lower boundary of the range) is stored in the DOC data setting register 0 (DODSR0) and 16-bit/32-bit data value 2 (upper boundary of the range) is stored in the DOC data setting register 1 (DODSR1)

[Syntax]

void R_<Config_DOC>_SetMode (doc_mode_t *mode*, type *value1, type value 2*);

[Argument(s)]

I/O	Argument	Description
I	doc_mode_t *mode*;	Operating mode type (including the condition for detection) COMPARE_NEQ: Data comparison mode (not equal to) COMPARE_EQ: Data comparison mode (equal to) COMPARE_GT: Data comparison mode (greater than) COMPARE_LT: Data comparison mode (less than) COMPARE_IN_RANGE: Data comparison mode (within the range) COMPARE_OUT_RANGE: Data comparison mode (beyond of range) ADDITION: Data addition mode SUBTRACTION: Data subtraction mode
I	type *value1*;	- Reference value for compare operation except range comparison, operation result for addition or subtraction - Low boundary of the range for range comparison - The “type” can be “uint16_t” or “uint32_t”
I	type *value2*;	- Upper boundary of the range for range comparison - The “type” can be “uint16_t” or “uint32_t”

[Return value]

None.

R_<Config_DOC>_WriteData

This API function sets the input value (value for comparison with, addition to, or subtraction from the reference value) for use in the operation.

[Syntax]

```
void        R_<Config_DOC>_WriteData (type data);
```

[Argument(s)]

I/O	Argument	Description
I	type *data*;	Input value for use in the operation, the “type” can be “uint16_t” or “uint32_t”

[Return value]

None.

R_<Config_DOC>_GetResult

This API function gets the result of the operation.

[Syntax]

```
void        R_<Config_DOC>_GetResult ( type * const data );
```

[Argument(s)]

I/O	Argument	Description
O	type * const *data*;	Pointer to the location where the result of the operation is to be stored; the “type” can be “uint16_t” or “uint32_t”

[Return value]

None.

R_<Config_DOC>_ClearFlag

This API function clears the data operation circuit flag.

[Syntax]

```
void        R_<Config_DOC>_ClearFlag ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DOC>_Create_UserInit

This API function executes user-specific initialization processing for the data operation circuit.

Remark This API function is called from R_Config_DOC_Create as a callback routine.

[Syntax]

```
void R_<Config_DOC>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_DOC>_dopcf_interrupt

r_<Config_DOC>_dopci_interrupt

This API function executes processing in response to data operation circuit interrupts.

Remark This API function is called to run interrupt processing in response to the data operation circuit interrupt, which is generated when the result of data comparison satisfies the condition for detection, the result of data addition is greater than 0xFFFF, or the result of data subtraction is less than 0x0.

[Syntax]

```
void    r_<Config_DOC>_dopcf_interrupt ( void );
```

```
void    r_<Config_DOC>_dopci_interrupt ( void );
```

Remark The name of this API function varies with the device group.

[Argument(s)]

None.

[Return value]

None.

Usage example

Adding an array of values in data addition mode and getting the result of addition in response to the interrupt when it has exceeded "FFFFh";

changing the operating mode to data comparison mismatch mode and generating an interrupt when a value other than "000h" is detected in the array:

main.c

```
#include "r_smc_entry.h"
extern volatile uint16_t data[16];
void main(void)
{
    uint8_t cnt;
    while (1U)
    {
        for (cnt = 0; cnt < 16U; cnt++)
        {
            /* Write new data to compare */
            R_<Config_DOC>_WriteData(data[cnt]);
        }
    }
}
```

Config_DOC_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t data[16];
volatile uint16_t result;
/* End user code. Do not edit comment generated here */

void r_<Config_DOC>_dopci_interrupt(void)
{
    /* Start user code for r_<Config_DOC_dopci_interrupt. Do not edit comment generated here */
    /* Get result */
    R_<Config_DOC>_GetResult((uint16_t *)&result);

    /* Configure the operation mode of DOC */
    R_<Config_DOC>_SetMode(COMPARE_MISMATCH, 0x0000);

    /* Clear DOPCI flag */
    R_<Config_DOC>_ClearFlag();
    /* End user code. Do not edit comment generated here */
}
```

4.2.12 Data Transfer Controller

The Code Generator outputs the following API functions for the data transfer controller.

Table4.12 API Functions: [Data Transfer Controller]

API Function Name	Function
R_<Config_DTC>_Create	Executes initialization processing that is required before controlling the data transfer controller.
R_<Config_DTC>_Start	Enables activation of the data transfer controller.
R_<Config_DTC>_Stop	Disables activation of the data transfer controller.
R_<Config_DTC>_Create_UserInit	Executes user-specific initialization processing for the data transfer controller.

R_<Config_DTC>_Create

This API function executes initialization processing that is required before controlling the data operation circuit.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_DTC>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DTC>_Start

This API function sets the operating mode and the initial reference value for use by the data operation circuit.

Remark This API function manipulates the DTCE bit corresponding to the selected activation source to enable activation of the data transfer controller.

[Syntax]

```
void R_<Config_DTC>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DTC>_Stop

This API function disables activation of the data transfer controller.

Remark This API function manipulates the DTCE bit corresponding to the selected activation source to disable activation of the data transfer controller.

[Syntax]

```
void    R_<Config_DTC>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

4.2.12.1 R_<Config_DTC>_Create_UserInit

This API function executes user-specific initialization processing for the data transfer controller.

Remark This API function is called from R_<Config_DTC>_Create as a callback routine.

[Syntax]

```
void R_<Config_DTC>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

4.2.12.2 Usage example

Starting DTC data transfer in response to a compare match interrupt:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start CMT channel 0 counter */
    R_Config_CMT0_Start();

    /* Enable operation of transfer data DTC */
    R_Config_DTC_Start();

    while (1U)
    {
        nop();
    }
}
```

4.2.13 Dead Time Compensaion Counter

The Code Generator outputs the following API functions for the dead time compensation counter.

Table4.13 API Functions: [Dead Time Compensation Counter]

API Function Name	Function
R_<Config_MTU5>_Create	Executes initialization processing that is required before controlling the dead time compensation counter.
R_<Config_MTU5>_U5_Start	Starts U phase counting by the counter.
R_<Config_MTU5>_U5_Stop	Stops U phase counting by the counter.
R_<Config_MTU5>_V5_Start	Starts V phase counting by the counter
R_<Config_MTU5>_V5_Stop	Stops V phase counting by the counter
R_<Config_MTU5>_W5_Start	Starts W phase counting by the counter
R_<Config_MTU5>_W5_Stop	Stops W phase counting by the counter
R_<Config_MTU5>_Create_UserInit	Executes user-specific initialization processing for the dead time compensation counter.
r_<Config_MTU5>_tgimn_interrupt	Executes processing in response to input capture interrupts.

R_<Config_MTU5>_Create

This API function executes initialization processing that is required before controlling the dead time compensation counter.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_MTU5>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU5>_U5_Start

This API function starts U phase counting by the counter.

[Syntax]

```
void        R_<Config_MTU5>_U5_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU5>_U5_Stop

This API function stops U phase counting by the counter.

[Syntax]

```
void        R_<Config_MTU5>_U5_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU5>_V5_Start

This API function starts V phase counting by the counter.

[Syntax]

```
void        R_<Config_MTU5>_V5_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU5>_V5_Stop

This API function stops V phase counting by the counter.

[Syntax]

```
void        R_<Config_MTU5>_V5_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU5>_W5_Start

This API function starts W phase counting by the counter.

[Syntax]

```
void        R_<Config_MTU5>_W5_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU5>_W5_Stop

This API function stops W phase counting by the counter.

[Syntax]

```
void        R_<Config_MTU5>_W5_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU5>_Create_UserInit

This API function executes user-specific initialization processing for the dead time compensation counter.

Remark This API function is called from R_<Config_MTU5>_Create as a callback routine.

[Syntax]

```
void R_<Config_MTU5>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU5>_tgi*mn*_interrupt

This API function executes processing in response to input capture interrupts.

Remark This API function is called as the interrupt handler for input capture interrupts, which occur when the selected edge is detected on the input signal line.

[Syntax]

```
void    r_<Config_MTU5>_tgimn_interrupt ( void );
```

Remark *n* is a channel number and *m* is the number of a timer general register.

[Argument(s)]

None.

[Return value]

None.

Usage example

Compensating for the dead time in PWM waveform output when MTU6 and MTU7 are being used in complementary PWM mode:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the MTU5 channel counter */
    R_Config_MTU5_Start();

    /* Start the MTU6 channel counter */
    R_Config_MTU6_MTU7_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_MTU5_user.c

```
static void r_<Config_MTU5_tgiu5_interrupt(void)
{
    /* Start user code for r_<Config_MTU5_tgiu5_interrupt. Do not edit comment generated here */
    /* Write the corrected value */
    if ( MTU6.TGRB > MTU5.TGRU )
    {
        MTU6.TGRD = (MTU6.TGRB - MTU5.TGRU);
    }
    /* End user code. Do not edit comment generated here */
}
```

4.2.14 DMA Controller

The Code Generator outputs the following API functions for the DMA controller.

Table4.14 API Functions: [DMA Controller]

API Function Name	Function
R_<Config_DMAC0>_Create	Executes initialization processing that is required before controlling the DMA controller.
R_<Config_DMAC0>_Start	Starts the DMAC waiting for DMA activation triggers.
R_<Config_DMAC0>_Stop	Stops the DMAC waiting for DMA activation triggers.
R_<Config_DMAC0>_Set_SoftwareTrigger	Sets a software transfer request.
R_<Config_DMAC0>_Clear_SoftwareTrigger	Clears a software transfer request.
R_<Config_DMAC0>_Create_UserInit	Executes user-specific initialization processing for the DMA controller.
r_<Config_DMAC0>_dmacni_interrupt	Executes processing in response to transfer end interrupts from channel *n*. (*n* = 0 to 3)
r_dmacn_callback_transfer_end	Executes processing in response to transfer end interrupts from channel *n*. (*n* = 0 to 3)
r_dmacn_callback_transfer_escape_end	Executes processing in response to transfer escape end interrupts from channel *n*. (*n* = 0 to 3)
r_dmac_dmac74i_interrupt	Executes processing in response to transfer end interrupts from channels 4 to 7.
r_dmacn_callback_transfer_end	Executes processing in response to transfer end interrupts from channel *n*. (*n* = 4 to 7)
r_dmacn_callback_transfer_escape_end	Executes processing in response to transfer escape end interrupts from channel *n*. (*n* = 4 to 7)

R_<Config_DMAC0>_Create

This API function executes initialization processing that is required before controlling the DMA controller.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_DMAC0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DMAC0>_Start

This API function starts the DMAC waiting for DMA activation triggers.

[Syntax]

```
void        R_<Config_DMAC0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DMAC0>_Stop

This API function stops the DMAC waiting for DMA activation triggers.

[Syntax]

```
void        R_<Config_DMAC0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DMAC0>_Set_SoftwareTrigger

This API function sets a software transfer request.

Remark When using software trigger continuously, at first, set the CLRS bit to 1 directly. After that, use this function to set SWREQ bit to 1. Finally, call R_<Config_DAMC0>_Clear_SoftwareTrigger to clear SWREQ bit when transferring operation is completed.

[Syntax]

```
void    R_<Config_DMAC0>_Set_SoftwareTrigger ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DMAC0>_Clear_SoftwareTrigger

This API function clears a software transfer request.

[Syntax]

```
void        R_<Config_DMAC0>_Clear_SoftwareTrigger ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_DMAC0>_Create_UserInit

This API function executes user-specific initialization processing for the DMA controller.

Remark This API function is called from R_<Config_DMAC0>_Create as a callback routine.

[Syntax]

```
void R_<Config_DMAC0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_DMAC0>_dmac*n*i_interrupt

This API function executes processing in response to transfer end interrupts from channel n.

[Syntax]

```
void        r_<Config_DMAC0>_dmacni_interrupt ( void );
```

Remark *n* is a channel number. (*n* = 0 to 3)

[Argument(s)]

None.

[Return value]

None.

r_dmac*n*_callback_transfer_end

This API function executes processing in response to transfer end interrupts from channel n.

Remark This API function is called as a callback routine from r_<Config_DMAC0>_dmacni_interrupt, which is the interrupt handler for transfer end interrupts from channel n.

[Syntax]

```
void r_dmacn_callback_transfer_end ( void );
```

Remark *n* is a channel number. (*n* = 0 to 3)

[Argument(s)]

None.

[Return value]

None.

r_dmac*n*_callback_transfer_escape_end

This API function executes processing in response to transfer escape end interrupts from channel *n*.

Remark This API function is called as a callback routine from r_<Config_DMAC0>_dmacni_interrupt, which is the interrupt handler for transfer escape end interrupts from channel *n*.

[Syntax]

```
void r_dmacn_callback_transfer_escape_end ( void );
```

Remark *n* is a channel number. (*n* = 0 to 3)

[Argument(s)]

None.

[Return value]

None.

r_dmac_dmac74i_interrupt

This API function executes processing in response to transfer end interrupts from channels 4 to 7.

[Syntax]

```
void    r_dmac_dmac74i_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_dmac*n*_callback_transfer_end

This API function executes processing in response to transfer end interrupts from channel n.

Remark This API function is called as a callback routine from r_dmac_dmac74i_interrupt, which is the interrupt handler for transfer end interrupts from channel *n*.

[Syntax]

```
void r_dmacn_callback_transfer_end ( void );
```

Remark *n* is a channel number. (*n* = 4 to 7)

[Argument(s)]

None.

[Return value]

None.

r_dmac*n*_callback_transfer_escape_end

This API function executes processing in response to transfer escape end interrupts from channel n.

Remark This API function is called as a callback routine from r_dmac_dmac74i_interrupt, which is the interrupt handler for transfer escape end interrupts from channel n.

[Syntax]

```
void r_dmacn_callback_transfer_escape_end ( void );
```

Remark *n* is a channel number. (*n* = 4 to 7)

[Argument(s)]

None.

[Return value]

None.

Usage example

Starting a transfer in response to a compare match interrupt, and setting a flag when the transfer is completed:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start CMT channel 0 counter */
    R_Config_CMT0_Start();

    /* Enable the DMAC0 activation */
    R_Config_DMAC0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_DMAC0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_dmac0_f;
/* End user code. Do not edit comment generated here */

void R_Config_DMAC0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    /* Clear the flag */
    g_dmac0_f = 0U;
    /* End user code. Do not edit comment generated here */
}

static void r_dmac0_callback_transfer_end(void)
{
    /* Start user code for r_dmac0_callback_transfer_end. Do not edit comment generated here */
    /* Set the flag */
    g_dmac0_f = 1U;
    /* End user code. Do not edit comment generated here */
}
```

Config_DMAC0.h

```
#define _DMAC0_ACTIVATION_SOURCE      (28U) /* Please assign dynamic vector in interrupt tab */
```

Note: This line will be lost when the code is re-generated, so be sure to write it again after re-generating the code.

4.2.15 Event Link Controller

The Code Generator outputs the following API functions for the event link controller.

Table4.15 API Functions: [Event Link Controller]

API Function Name	Function
R_<Config_ELC>_Create	Executes initialization processing that is required before controlling the event link controller.
R_<Config_ELC>_Start	Starts interlinked operation of peripheral functions.
R_<Config_ELC>_Stop	Stops interlinked operation of peripheral functions.
R_<Config_ELC>_GenerateSoftwareEvent	Generates a software event.
R_<Config_ELC>_Set_PortBuffern	Sets the value of a port buffer.
R_<Config_ELC>_Get_PortBuffern	Gets the value of a port buffer.
R_<Config_ELC>_Create_UserInit	Executes user-specific initialization processing for the event link controller.
r_<Config_ELC>_elsrni_interrupt	Executes processing in response to event link interrupts.

R_<Config_ELC>_Create

This API function executes initialization processing that is required before controlling the event link controller.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_ELC>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ELC>_Start

This API function starts interlinked operation of peripheral functions.

[Syntax]

```
void        R_<Config_ELC>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ELC>_Stop

This API function stops interlinked operation of peripheral functions.

[Syntax]

```
void        R_<Config_ELC>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ELC>_GenerateSoftwareEvent

This API function generates a software event.

[Syntax]

```
void        R_<Config_ELC>_GenerateSoftwareEvent ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ELC>_Set_PortBuffer*n*

This API function sets the value of a port buffer.

[Syntax]

```
void        R_<Config_ELC>_Set_PortBuffer n ( uint8_t value );
```

Remark *n* is a port number.

[Argument(s)]

I/O	Argument	Description
I	uint8_t *value*;	Value to be written to the port buffer

[Return value]

None.

R_<Config_ELC>_Get_PortBuffer*n*

This API function gets the value of a port buffer.

[Syntax]

void R_<Config_ELC>_Get_PortBuffer*n* (uint8_t * const *value*);

Remark *n* is a port number.

[Argument(s)]

I/O	Argument	Description
O	uint8_t * const *value*;	Pointer to the location where the read value is to be stored

[Return value]

None.

R_<Config_ELC>_Create_UserInit

This API function executes user-specific initialization processing for the event link controller.

Remark This API function is called from R_<Config_ELC>_Create as a callback routine.

[Syntax]

```
void R_<Config_ELC>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_ELC>_elsr*n*i_interrupt

This API function executes processing in response to event link interrupts.

[Syntax]

```
void        r_<Config_ELC>_elsrni_interrupt ( void );
```

Remark *n* is the number of an event link setting register.

[Argument(s)]

None.

[Return value]

None.

Usage example

Generating a software event and generating linked events;
linking events one after another and terminating execution in response to an event link interrupt:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Enable all ELC event links */
    R_Config_ELC_Start();

    /* Trigger a software event */
    R_Config_ELC_GenerateSoftwareEvent();

    while (1U)
    {
        nop();
    }
}
```

Config_ELC_user.c

```
static void r_Config_ELC_elsr18i_interrupt(void)
{
    /* Start user code for r_Config_ELC_elsr18i_interrupt. Do not edit comment generated here */
    /* Disable all ELC event links */
    R_Config_ELC_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.16 General PWM Timer (GPT)

The Code Generator outputs the following API functions for the general PWM timer (GPT).

Table4.16 API Functions: [General PWM Timer (GPT)]

API Function Name	Function
R_<Config_GPT0>_Create	Executes initialization processing that is required before controlling the general PWM timer (GPT).
R_<Config_GPT0>_Start	Starts counting by the counter.
R_<Config_GPT0>_Stop	Stops counting by the counter.
R_<Config_GPT0>_HardwareStart	Enables interrupts.
R_<Config_GPT0>_HardwareStop	Disables interrupts.
R_<Config_GPT0>_ETGI_Start	Enables interrupts due to the input of a falling-edge or rising-edge external trigger.
R_<Config_GPT0>_ETGI_Stop	Disables interrupts due to the input of a falling-edge or rising-edge external trigger.
R_<Config_GPT0>_Software_Clear	Clears the counter for the general PWM timer (GPT).
R_<Config_GPT0>_Create_UserInit	Executes user-specific initialization processing for the general PWM timer (GPT).
r_<Config_GPT0>_gtcimn_interrupt	Executes processing in response to input capture interrupts or compare match interrupts.
r_<Config_GPT0>_gtcivn_interrupt	Executes processing in response to overflow interrupts.
r_<Config_GPT0>_gtciun_interrupt	Executes processing in response to underflow interrupts.
r_<Config_GPT0>_gdten_interrupt	Executes processing in response to dead time error interrupts.
r_gpt_etgin_interrupt	Executes processing in response to interrupts due to the input of a falling-edge external trigger.
r_gpt_etgip_interrupt	Executes processing in response to interrupts due to the input of a rising-edge external trigger.

R_<Config_GPT0>_Create

This API function executes initialization processing that is required before controlling the general PWM timer (GPT).

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_GPT0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_GPT0>_Start

This API function starts counting by the counter.

Remark When using RX260 / RX261 / RX26T / RX66T / RX66N / RX72N / RX72M / RX72T, this function is empty if Count start sources setting is not enabled, and interrupt is not configured for GPT.

[Syntax]

```
void    R_<Config_GPT0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_GPT0>_Stop

This API function stops counting by the counter.

Remark When using RX260 / RX261 / RX26T / RX66T / RX66N / RX72N / RX72M / RX72T, this function is empty if Count stop sources setting is not enabled, and interrupt is not configured for GPT.

[Syntax]

```
void        R_<Config_GPT0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_GPT0>_HardwareStart

This API function enables interrupts.

Remark This API function is used to enable the detection of interrupts during counting that is started by an external or internal trigger (hardware source).

[Syntax]

```
void    R_<Config_GPT0>_HardwareStart ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_GPT0>_HardwareStop

This API function disables interrupts.

Remark This API function is used to disable the detection of interrupts during counting that is started by an external or internal trigger (hardware source).

[Syntax]

```
void    R_<Config_GPT0>_HardwareStop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_GPT0>_ETGI_Start

This API function enables interrupts due to the input of a falling-edge or rising-edge external trigger.

[Syntax]

```
void        R_<Config_GPT0>_ETGI_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_GPT0>_ETGI_Stop

This API function disables interrupts due to the input of a falling-edge or rising-edge external trigger.

[Syntax]

```
void        R_<Config_GPT0>_ETGI_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_GPT0>_Software_Clear

This API function clears the counter for the general PWM timer (GPT).

[Syntax]

```
void        R_<Config_GPT0>_Software_Clear ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_GPT0>_Create_UserInit

This API function executes user-specific initialization processing for the general PWM timer (GPT).

Remark This API function is called from R_<Config_GPT0>_Create as a callback routine.

[Syntax]

```
void R_<Config_GPT0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_GPT0>_gtci*mn*_interrupt

This API function executes processing in response to input capture interrupts or compare match interrupts.

[Syntax]

```
void        r_<Config_GPT0>_gtci mn _interrupt ( void );
```

Remark *n* is a channel number and *m* is the number of a timer general register.

[Argument(s)]

None.

[Return value]

None.

r_<Config_GPT0>_gtciv*n*_interrupt

This API function executes processing in response to overflow interrupts.

[Syntax]

```
void        r_<Config_GPT0>_gtciv n_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

r_<Config_GPT0>_gtciu*n*_interrupt

This API function executes processing in response to underflow interrupts.

[Syntax]

```
void        r_<Config_GPT0>_gtciun_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

r_<Config_GPT0>_gdte*n*_interrupt

This API function executes processing in response to dead time error interrupts.

[Syntax]

```
void        r_<Config_GPT0>_gdten_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

r_gpt_etgin_interrupt

This API function executes processing in response to interrupts due to the input of a falling-edge external trigger.

[Syntax]

```
void    r_gpt_etgin_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_gpt_etgip_interrupt

This API function executes processing in response to interrupts due to the input of a rising-edge external trigger.

[Syntax]

```
void    r_gpt_etgip_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Acquiring a captured value:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start GPT channel 0 counter */
    R_Config_GPT0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_GPT0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t g_gpt0_capture_value;
/* End user code. Do not edit comment generated here */

static void r_Config_GPT0_gtcia0_interrupt(void)
{
    /* Start user code for r_Config_GPT0_gtcia0_interrupt. Do not edit comment generated here */
    g_gpt0_capture_value = GPT0.GTCCRA;
    /* End user code. Do not edit comment generated here */
}
```

4.2.17 Group Scan Mode S12AD

The Code Generator outputs the following API functions for the group scan mode S12AD.

Table4.17 API Functions: [Group Scan Mode S12AD]

API Function Name	Function
R_<Config_S12AD0>_Create	Executes initialization processing that is required before controlling the S12AD in group scan mode.
R_<Config_S12AD0>_Start	Starts A/D conversion.
R_<Config_S12AD0>_Stop	Stops A/D conversion.
R_<Config_S12AD0>_Get_ValueResult	Gets the result of conversion.
R_<Config_S12AD0>_Set_CompareValue	Sets the compare level.
R_<Config_S12AD0>_Set_CompareAValue	Sets the compare level for window A.
R_<Config_S12AD0>_Set_CompareBValue	Sets the compare level for window B.
R_<Config_S12AD0>_Create_UserInit	Executes user-specific initialization processing for the S12AD in group scan mode.
r_<Config_S12AD0>_interrupt	Executes processing in response to scan end interrupts.
r_<Config_S12AD0>_compare_interrupt	Executes processing in response to compare interrupts.
r_<Config_S12AD0>_compare_interruptA	Executes processing in response to compare interrupts for window A.
r_<Config_S12AD0>_compare_interruptB	Executes processing in response to compare interrupts for window B.
r_<Config_S12AD0>_groupb_interrupt	Executes processing in response to scan end interrupts for group B.
r_<Config_S12AD0>_groupc_interrupt	Executes processing in response to scan end interrupts for group C.

R_<Config_S12AD0>_Create

This API function executes initialization processing that is required before controlling the group scan mode S12AD.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_S12AD0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Start

This API function starts A/D conversion.

[Syntax]

```
void        R_<Config_S12AD0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Stop

This API function stops A/D conversion.

[Syntax]

```
void        R_<Config_S12AD0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Get_ValueResult

This API function gets the result of conversion.

[Syntax]

```
void    R_<Config_S12AD0>_Get_ValueResult ( ad_channel_t channel, uint16_t * const buffer );
```

[Argument(s)]

For RX130 or RX230/RX231

I/O	Argument	Description
I	ad_channel_t *channel*;	Channel number ADCHANNEL0: Input channel AN000 ADCHANNEL1: Input channel AN001 ADCHANNEL2: Input channel AN002 ADCHANNEL3: Input channel AN003 ADCHANNEL4: Input channel AN004 ADCHANNEL5: Input channel AN005 ADCHANNEL6: Input channel AN006 ADCHANNEL7: Input channel AN007 ADCHANNEL16: Input channel AN016 ADCHANNEL17: Input channel AN017 ADCHANNEL18: Input channel AN018 ADCHANNEL19: Input channel AN019 ADCHANNEL20: Input channel AN020 ADCHANNEL21: Input channel AN021 ADCHANNEL22: Input channel AN022 ADCHANNEL23: Input channel AN023 ADCHANNEL24: Input channel AN024 ADCHANNEL25: Input channel AN025 ADCHANNEL26: Input channel AN026 ADCHANNEL27: Input channel AN027 ADCHANNEL28: Input channel AN028 ADCHANNEL29: Input channel AN029 ADCHANNEL30: Input channel AN030 ADCHANNEL31: Input channel AN031 ADTEMPSENSOR: Extended analog input (temperature sensor output) ADINTERREFVOLT: Extended analog input (internal reference voltage) ADSELFDIAGNOSIS: Result of self-diagnosis ADDATADUPLICATION: Double-trigger mode result
O	uint16_t * const *buffer*;	Pointer to the location where the acquired results of conversion are to be stored

For other devices

I/O	Argument	Description
I	ad_channel_t *channel*;	Channel number ADCHANNEL0: Input channel AN000 ADCHANNEL1: Input channel AN001 ADCHANNEL2: Input channel AN002 ADCHANNEL3: Input channel AN003 ADCHANNEL4: Input channel AN004 ADCHANNEL5: Input channel AN005 ADCHANNEL6: Input channel AN006 ADCHANNEL7: Input channel AN007 ADCHANNEL8: Input channel AN008 ADCHANNEL9: Input channel AN009 ADCHANNEL10: Input channel AN010 ADCHANNEL11: Input channel AN011 ADCHANNEL12: Input channel AN012 ADCHANNEL13: Input channel AN013 ADCHANNEL14: Input channel AN014 ADCHANNEL15: Input channel AN015 ADCHANNEL16: Input channel AN016 ADCHANNEL17: Input channel AN017 ADCHANNEL18: Input channel AN018 ADCHANNEL19: Input channel AN019 ADCHANNEL20: Input channel AN020 ADTEMPSENSOR: Extended analog input (temperature sensor output) ADINTERREFVOLT: Extended analog input (internal reference voltage) ADSELFDIAGNOSIS: Result of self-diagnosis ADDATADUPLICATION: Double-trigger mode result ADDATADUPLICATIONA: Double-trigger mode A result ADDATADUPLICATIONB: Double-trigger mode B result
O	uint16_t * const *buffer*;	Pointer to the location where the acquired results of conversion are to be stored

[Return value]

None.

R_<Config_S12AD0>_Set_CompareValue

This API function sets the compare level.

[Syntax]

```
void    R_<Config_S12AD0>_Set_CompareValue ( uint16_t reg_value0, uint16_t reg_value1);
```

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Set_CompareAValue

This API function sets the compare level for window A.

[Syntax]

```
void    R_<Config_S12AD0>_Set_CompareAValue ( uint16_t reg_value0, uint16_t reg_value1);
```

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Set_CompareBValue

This API function sets the compare level for window B.

[Syntax]

void R_<Config_S12AD0>_Set_CompareBValue (uint16_t *reg_value0*, uint16_t *reg_value1*);

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Create_UserInit

This API function executes user-specific initialization processing for the group scan mode S12AD.

Remark This API function is called from R_<Config_S12AD0>_Create as a callback routine.

[Syntax]

```
void R_<Config_S12AD0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_interrupt

This API function executes processing in response to scan end interrupts.

[Syntax]

```
void        r_<Config_S12AD0>_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interrupt

This API function executes processing in response to compare interrupts.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interruptA

This API function executes processing in response to compare interrupts for window A.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interruptA ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interruptB

This API function executes processing in response to compare interrupts for window B.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interruptB ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_groupb_interrupt

This API function executes processing in response to scan end interrupts for group B.

[Syntax]

```
void        r_<Config_S12AD0>_groupb_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_groupc_interrupt

This API function executes processing in response to scan end interrupts for group C.

[Syntax]

```
void        r_<Config_S12AD0>_groupc_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Getting the result of A/D conversions from groups A and B:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the AD0 converter */
    R_Config_S12AD0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_S12AD0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t g_s12ad0_ch000_value;
volatile uint16_t g_s12ad0_ch001_value;
/* End user code. Do not edit comment generated here */

static void r_Config_S12AD0_interrupt(void)
{
    /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
    /* Get result from the AD0 channel 0 (AN000) converter */
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, (uint16_t *)&g_s12ad0_ch000_value);
    /* End user code. Do not edit comment generated here */
}

static void r_Config_S12AD0_groupb_interrupt(void)
{
    /* Start user code for r_Config_S12AD0_groupb_interrupt. Do not edit comment generated here
*/
    /* Get result from the AD0 channel 1 (AN001) converter */
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL1, (uint16_t *)&g_s12ad0_ch001_value);
    /* End user code. Do not edit comment generated here */
}
```

4.2.18 I2C Master Mode

The Code Generator outputs the following API functions for I2C communications in master mode (RIIC/SCI/RSCI).

Table 4.18 API Functions: [I2C Master Mode (RIIC)]

API Function Name	Function
R_<Config_RIIC0>_Create	Executes initialization processing that is required before controlling I2C communications in master mode.
R_<Config_RIIC0>_Start	Starts serial communications.
R_<Config_RIIC0>_Stop	Stops serial communications.
R_<Config_RIIC0>_Master_Send	Starts master transmission.
R_<Config_RIIC0>_Master_Send_Without_Stop	Starts master transmission (with no stop condition being issued at the end of transmission).
R_<Config_RIIC0>_Master_Receive	Starts master reception.
R_<Config_RIIC0>_IIC_StartCondition	Issues a start condition.
R_<Config_RIIC0>_IIC_StopCondition	Issues a stop condition.
r_<Config_RIIC0>_error_interrupt	Executes processing in response to communication error interrupts or communication event generation interrupts.
r_<Config_RIIC0>_receive_interrupt	Executes processing in response to receive data full interrupts.
r_<Config_RIIC0>_transmit_interrupt	Executes processing in response to transmit data empty interrupts.
r_<Config_RIIC0>_transmitend_interrupt	Executes processing in response to transmit end interrupts.
r_<Config_RIIC0>_callback_error	Executes processing specific to the detection of a loss in arbitration, NACK, timeout or communication sequence error among the sources of communication error interrupts or communication event generation interrupts.
r_<Config_RIIC0>_callback_transmitend	Executes processing specific to the detection of a stop condition in master transmission among the sources of communication error interrupts or communication event generation interrupts. In master transmission without issuing a stop condition, this API function executes processing in response to transmit end interrupts.
r_<Config_RIIC0>_callback_receiveend	Executes processing specific to the detection of a stop condition in master reception among the sources of communication error interrupts or communication event generation interrupts.

Table 4.19 API Functions: [I2C Master Mode (SCI/RSCI simple I2C mode)]

API Function Name	Function
R_<Config_SCI0>_Create	Executes initialization processing that is required before controlling I2C communications in master mode.
R_<Config_SCI0>_Start	Starts serial communications.
R_<Config_SCI0>_Stop	Stops serial communications.
R_<Config_SCI0>_IIC_Master_Send	Starts master transmission.
R_<Config_SCI0>_IIC_Master_Receive	Starts master reception.
R_<Config_SCI0>_IIC_StartCondition	Issues a start condition.
R_<Config_SCI0>_IIC_StopCondition	Issues a stop condition.
R_<Config_SCI0>_Create_UserInit	Executes user-specific initialization processing for I2C communications in master mode.
r_<Config_SCI0>_receive_interrupt	Executes processing in response to receive data full interrupts.
r_<Config_SCI0>_transmit_interrupt	Executes processing in response to transmit data empty interrupts.
r_<Config_SCI0>_transmitend_interrupt	Executes processing in response to interrupts on completion of generation of a start condition/restart condition/stop condition.
r_<Config_SCI0>_callback_transmitend	Executes processing specific to the detection of a stop condition in master transmission among the sources of interrupts on completion of generation of a start condition/restart condition/stop condition. When the transmit data empty interrupt is selected as a DTC or DMAC activation source, this API function executes processing in response to the interrupt.
r_<Config_SCI0>_callback_receiveend	Executes processing specific to the detection of a stop condition in master reception among the sources of interrupts on completion of generation of a start condition/restart condition/stop condition. When the receive data full interrupt is selected as a DTC or DMAC activation source, this API function executes processing in response to the interrupt.

R_<Config_RIIC0>_Create

This API function executes initialization processing that is required before controlling I2C communications in master mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_RIIC0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RIIC0>_Start

This API function starts serial communications.

[Syntax]

```
void        R_<Config_RIIC0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RIIC0>_Stop

This API function stops serial communications.

[Syntax]

```
void        R_<Config_RIIC0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RIIC0>_Master_Send

This API function starts master transmission.

Remark1. This API function executes the master transmission of the slave address specified by the argument *adr* and the R/W# bit to slave devices, and then repeats the master transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. This API function internally calls R_<Config_RIIC0>_IIC_StartCondition to start master transmission.

Remark3. A stop condition is issued in r_<Config_RIIC0>_transmitend_interrupt to stop master transmission.

Remark4. Calling R_<Config_RIIC0>_Start is required before this API function is called to execute master transmission.

[Syntax]

```
MD_STATUS  R_<Config_RIIC0>_Master_Send ( uint16_t adr, uint8_t * const tx_buf, uint16_t tx_num );
```

[Argument(s)]

I/O	Argument	Description
I	uint16_t *adr*;	Slave address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Slave address (0 to 1023)
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR1	The bus is busy.
MD_ERROR2	The specification of argument *adr* is invalid.

R_<Config_RIIC0>_Master_Send_Without_Stop

his API function starts master transmission (with no stop condition being issued at the end of transmission).

Remark1. This API function executes the master transmission of the slave address specified by the argument *adr* and the R/W# bit to slave devices, and then repeats the master transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. This API function internally calls R_<Config_RIIC0>_IIC_StartCondition to start master transmission.

Remark3. r_<Config_RIIC0>_transmitend_interrupt does not issue a stop condition to stop master transmission.

Remark4. Calling R_<Config_RIIC0>_Start is required before this API function is called to execute master transmission.

[Syntax]

```
MD_STATUS R_<Config_RIIC0>_Master_Send_Without_Stop ( uint16_t adr, uint8_t * const tx_buf, uint16_t tx_num );
```

[Argument(s)]

I/O	Argument	Description
I	uint16_t *adr*;	Slave address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Slave address (0 to 1023)
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR1	The bus is busy.
MD_ERROR2	The specification of argument *adr* is invalid.

R_<Config_RIIC0>_Master_Receive

This API function starts master reception.

Remark1. This API function executes the master transmission of the slave address specified by the argument *adr* and the R/W# bit to slave devices, and then repeats the master reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark2. This API function internally calls R_<Config_RIIC0>_StartCondition to start master reception.

Remark3. A stop condition is issued in r_<Config_RIIC0>_receive_interrupt to stop master reception.

Remark4. Calling R_<Config_RIIC0>_Start is required before this API function is called to execute master reception.

[Syntax]

```
MD_STATUS    R_<Config_RIIC0>_Master_Receive ( uint16_t adr, uint8_t * const rx_buf, uint16_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
I	uint16_t *adr*;	Slave address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Slave address (0 to 127)
O	uint8_t * const *rx_buf*;	Pointer to the buffer where the received data are to be stored
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR2	The specification of argument *adr* is invalid.
MD_ERROR3	The specification of argument *adr* is invalid (10-bit addresses are not supported in master reception).
MD_ERROR4	The bus is busy (timeout or loss in arbitration has been detected).
MD_ERROR5	The bus is busy (no stop condition has been detected).

R_<Config_RIIC0>_IIC_StartCondition

This API function issues a start condition.

Remark A call of this API function generates a communication error/communication event generation interrupt, after which r_<Config_RIIC0>_error_interrupt is called.

[Syntax]

```
void    R_<Config_RIIC0>_IIC_StartCondition ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RIIC0>_IIC_StopCondition

This API function issues a stop condition.

Remark A call of this API function generates a communication error/communication event generation interrupt, after which r_<Config_RIIC0>_error_interrupt is called.

[Syntax]

```
void R_<Config_RIIC0>_IIC_StopCondition ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RIIC0>_Create_UserInit

This API function executes user-specific initialization processing for I2C communications in master mode.

Remark This API function is called from R_<Config_RIIC0>_Create as a callback routine.

[Syntax]

```
void R_<Config_RIIC0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_error_interrupt

This API function executes processing in response to communication error interrupts or communication event generation interrupts.

Remark This API function is called as the interrupt handler for communication error interrupts or communication event generation interrupts (due to detection of a loss in arbitration, NACK, timeout, start condition, or stop condition).

[Syntax]

```
void    r_<Config_RIIC0>_error_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_receive_interrupt

This API function executes processing in response to receive data full interrupts.

[Syntax]

```
void        r_<Config_RIIC0>_receive_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_transmit_interrupt

This API function executes processing in response to transmit data empty interrupts.

[Syntax]

```
void        r_<Config_RIIC0>_transmit_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_transmitend_interrupt

This API function executes processing in response to transmit end interrupts.

[Syntax]

```
void        r_<Config_RIIC0>_transmitend_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_callback_error

This API function executes processing specific to the detection of a loss in arbitration, NACK, timeout or communication sequence error among the sources of communication error interrupts or communication event generation interrupts.

Remark This API function is called from r_<Config_RIIC0>_error_interrupt as a callback routine.

[Syntax]

```
void r_<Config_RIIC0>_callback_error ( MD_STATUS status );
```

[Argument(s)]

I/O	Argument	Description
I	MD_STATUS *status*;	Interrupt sources MD_ERROR1: Detection of loss in arbitration MD_ERROR2: Detection of timeout MD_ERROR3: Detection of NACK MD_ERROR4: Dectection of communication sequence error

[Return value]

None.

r_<Config_RIIC0>_callback_transmitend

This API function executes processing specific to the detection of a stop condition in master transmission among the sources of communication error interrupts or communication event generation interrupts.

Remark1. This API function is called from r_<Config_RIIC0>_error_interrupt as a callback routine.

Remark2. To execute master transmission, call R_<Config_RIIC0>_Master_Send.

In master transmission without issuing a stop condition, this API function executes processing in response to transmit end interrupts.

Remark3. This API function is called from r_<Config_RIIC0>_transmitend_interrupt as a callback routine.

Remark4. To execute master transmission, call R_<Config_RIIC0>_Master_Send_Without_Stop.

[Syntax]

```
void        r_<Config_RIIC0>_callback_transmitend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_callback_receiveend

This API function executes processing specific to the detection of a stop condition in master reception among the sources of communication error interrupts or communication event generation interrupts.

Remark1. This API function is called from r_<Config_RIIC0>_error_interrupt as a callback routine.

Remark2. To execute master reception, call R_<Config_RIIC0>_IIC_Master_Receive.

[Syntax]

```
void        r_<Config_RIIC0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Create

This API function executes initialization processing that is required before controlling I2C communications in master mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_<Config_SCI0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Start

This API function starts serial communications.

[Syntax]

```
void        R_<Config_SCI0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Stop

This API function stops serial communications.

[Syntax]

```
void        R_<Config_SCI0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_IIC_Master_Send

This API function starts master transmission.

Remark1. This API function executes the master transmission of the slave address specified by the argument *adr* and the R/W# bit to slave devices, and then repeats the master transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. This API function internally calls R_<Config_SCI0>_IIC_StartCondition to start master transmission.

Remark3. A stop condition is issued in r_<Config_SCI0>_transmit_interrupt to stop master transmission.

Remark4. Calling R_<Config_SCI0>_Start is required before this API function is called to execute master transmission.

[Syntax]

```
void    R_<Config_RIIC0>_IIC_Master_Send ( uint8_t adr, uint8_t * const tx_buf, uint16_t tx_num );
```

[Argument(s)]

I/O	Argument	Description
I	uint8_t *adr*;	Slave address 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 Slave Address (0 to 127) | W(0)
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

None.

R_<Config_SCI0>_IIC_Master_Receive

This API function starts master reception.

Remark1. This API function executes the master transmission of the slave address specified by the argument *adr* to slave devices, and then repeats the master reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark2. This API function internally calls R_<Config_SCI0>_IIC_StartCondition to start master reception.

Remark3. A stop condition is issued in r_<Config_SCI0>_receive_interrupt to stop master reception.

Remark4. Calling R_<Config_SCI0>_Start is required before this API function is called to execute master reception.

[Syntax]

```
void    R_<Config_SCI0>_IIC_Master_Receive ( uint8_t adr, uint8_t * const rx_buf, uint16_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
I	uint8_t *adr*;	Slave address 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 Slave Address (0 to 127) | R(1)
O	uint8_t * const *rx_buf*;	Pointer to the buffer where the received data are to be stored
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

None.

R_<Config_SCI0>_IIC_StartCondition

This API function issues a start condition.

Remark A call of this API function generates an interrupt on completion of generation of a start condition/restart condition/stop condition, after which r_<Config_SCI0>_transmitend_interrupt is called.

[Syntax]

```
void R_<Config_SCI0>_IIC_StartCondition ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_IIC_StopCondition

This API function issues a stop condition.

Remark A call of this API function generates an interrupt on completion of generation of a start condition/restart condition/stop condition, after which r_<Config_SCI0>_transmitend_interrupt is called.

[Syntax]

```
void R_<Config_SCI0>_IIC_StopCondition ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Create_UserInit

This API function executes user-specific initialization processing for I2C communications in master mode.

Remark This API function is called from R_<Config_SCI0>_Create as a callback routine.

[Syntax]

```
void R_<Config_SCI0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_receive_interrupt

This API function executes processing in response to receive data full interrupts.

[Syntax]

```
void        r_<Config_SCI0>_receive_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_transmit_interrupt

This API function executes processing in response to transmit data empty interrupts.

[Syntax]

```
void        r_<Config_SCI0>_transmit_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_transmitend_interrupt

This API function executes processing in response to interrupts on completion of generation of a start condition/restart condition/stop condition.

[Syntax]

```
void        r_<Config_SCI0>_transmitend_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_transmitend

This API function executes processing specific to the detection of a stop condition in master transmission among the sources of interrupts on completion of generation of a start condition/restart condition/stop condition.

Remark1. This API function is called from r_<Config_SCI0>_transmitend_interrupt as a callback routine.

Remark2. To execute master transmission, call R_<Config_SCI0>_IIC_Master_Send.

When the transmit data empty interrupt is selected as a DTC or DMAC activation source, this API function executes processing in response to the interrupt.

Remark3. his API function is called from r_<Config_SCI0>_transmit_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_SCI0>_callback_transmitend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_receiveend

This API function executes processing specific to the detection of a stop condition in master reception among the sources of interrupts on completion of generation of a start condition/restart condition/stop condition.

Remark1. This API function is called from r_Config_RIICn_transmitend_interrupt as a callback routine.

Remark2. To execute master reception, call R_<Config_SCI0>_IIC_Master_Receive.

When the receive data full interrupt is selected as a DTC or DMAC activation source, this API function executes processing in response to the interrupt.

Remark3. This API function is called from r_<Config_SCI0>_receive_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_SCI0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example 1

For RIIC, repeating master transmission four times:

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t g_riic0_tx_buf[2];
void main(void)
{
    /* Start the RIIC0 I2C Bus Interface */
    R_Config_RIIC0_Start();

    /* Send RIIC0 data to slave device [Slave address : 160 (10-bit address mode)] */
    R_Config_RIIC0_Master_Send(0x00A0, (uint8_t *)g_riic0_tx_buf, 2U);

    while (1U)
    {
        nop();
    }
}
```

Config_RIIC0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_riic0_tx_buf[2];
volatile uint8_t g_riic0_tx_cnt;
/* End user code. Do not edit comment generated here */

void R_Config_RIIC0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_riic0_tx_cnt = 0U;
    g_riic0_tx_buf[0] = g_riic0_tx_cnt;
    g_riic0_tx_buf[1] = 0x01;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_RIIC0_callback_transmitend(void)
{
    /* Start user code for r_Config_RIIC0_callback_transmitend. Do not edit comment generated
here */
    if ((++g_riic0_tx_cnt) < 4U)
    {
        g_riic0_tx_buf[0] = g_riic0_tx_cnt;
        g_riic0_tx_buf[1] += 0x01;

        /* Send RIIC0 data to slave device [Slave assress : 160 (10-bit address mode)] */
        R_Config_RIIC0_Master_Send(0x00A0, (uint8_t *)g_riic0_tx_buf, 2U);
    }
    else
    {
        /* Stop the RIIC0 I2C Bus Interface */
        R_Config_RIIC0_Stop();
    }
    /* End user code. Do not edit comment generated here */
}
```

Usage example 2

For SCI simple I2C mode, repeating master transmission four times:

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t g_sci0_tx_buf[2];
void main(void)
{
    /* Start the SCI0 I2C Bus Interface */
    R_Config_SCI0_Start();

    /* Send SCI0 data to slave device [Slave address : 80] */
    R_Config_SCI0_Master_Send(0xA0, (uint8_t *)g_sci0_tx_buf, 2U);

    while (1U)
    {
        nop();
    }
}
```

Config_SCI0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_sci0_tx_buf[2];
volatile uint8_t g_sci0_tx_cnt;
/* End user code. Do not edit comment generated here */

void R_Config_SCI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_sci0_tx_cnt = 0U;
    g_sci0_tx_buf[0] = g_riic0_tx_cnt;
    g_sco0_tx_buf[1] = 0x01;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI0_callback_transmitend. Do not edit comment generated
here */
    if ((++g_sci0_tx_cnt) < 4U)
    {
        g_sci0_tx_buf[0] = g_sci0_tx_cnt;
        g_sci0_tx_buf[1] += 0x01;

        /* Send SCI0 data to slave device [Slave assress : 80] */
        R_Config_SCI0_Master_Send(0xA0, (uint8_t *)g_sci0_tx_buf, 2U);
    }
    else
    {
        /* Stop the SCI0 I2C Bus Interface */
        R_Config_SCI0_Stop();
    }
    /* End user code. Do not edit comment generated here */
}
```

4.2.19 I2C Slave Mode

The Code Generator outputs the following API functions for I2C communications in slave mode.

Table4.20 API Functions: [I2C Slave Mode]

API Function Name	Function
R_<Config_RIIC0>_Create	Executes initialization processing that is required before controlling I2C communications in slave mode.
R_<Config_RIIC0>_Start	Starts serial communications.
R_<Config_RIIC0>_Stop	Stops serial communications.
R_<Config_RIIC0>_Slave_Send	Starts slave transmission.
R_<Config_RIIC0>_Slave_Receive	Starts slave reception.
R_<Config_RIIC0>_Create_UserInit	Executes user-specific initialization processing for I2C communications in slave mode.
r_<Config_RIIC0>_error_interrupt	Executes processing in response to communication error interrupts or communication event generation interrupts.
r_<Config_RIIC0>_receive_interrupt	Executes processing in response to receive data full interrupts.
r_<Config_RIIC0>_transmit_interrupt	Executes processing in response to transmit data empty interrupts.
r_<Config_RIIC0>_transmitend_interrupt	Executes processing in response to transmit end interrupts.
r_<Config_RIIC0>_callback_error	Executes processing specific to the detection of a loss in arbitration, NACK, timeout or communication sequence error among the sources of communication error interrupts or communication event generation interrupts.
r_<Config_RIIC0>_callback_transmitend	Executes processing specific to the detection of a stop condition in slave transmission among the sources of communication error interrupts or communication event generation interrupts.
r_<Config_RIIC0>_callback_receiveend	Executes processing specific to the detection of a stop condition in slave reception among the sources of communication error interrupts or communication event generation interrupts.

R_<Config_RIIC0>_Create

This API function executes initialization processing that is required before controlling I2C communications in slave mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_RIIC0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RIIC0>_Start

This API function starts serial communications.

[Syntax]

```
void        R_<Config_RIIC0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RIIC0>_Stop

This API function stops serial communications.

[Syntax]

```
void        R_<Config_RIIC0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RIIC0>_Slave_Send

This API function starts slave transmission.

Remark1. This API function repeats the slave transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. Calling R_<Config_RIIC0>_Start is required before this API function is called to execute transmission.

[Syntax]

```
MD_STATUS    R_<Config_RIIC0>_Slave_Send ( uint8_t * const tx_buf, uint16_t tx_num );
```

[Argument(s)]

I/O	Argument	Description
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end

R_<Config_RIIC0>_Slave_Receive

This API function starts slave reception.

Remark1. This API function repeats the slave reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark2. Calling R_<Config_RIIC0>_Start is required before this API function is called to execute slave reception.

[Syntax]

```
MD_STATUS      R_<Config_RIIC0>_Slave_Receive ( uint8_t * const rx_buf, uint16_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
O	uint8_t * const *rx_buf*;	Pointer to the buffer where the received data are to be stored
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end

R_<Config_RIIC0>_Create_UserInit

This API function executes user-specific initialization processing for I2C communications in slave mode.

Remark This API function is called from R_<Config_RIIC0>_Create as a callback routine.

[Syntax]

```
void R_<Config_RIIC0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_error_interrupt

This API function executes processing in response to communication error interrupts or communication event generation interrupts.

Remark This API function is called as the interrupt handler for communication error interrupts or communication event generation interrupts (due to detection of a loss in arbitration, NACK, timeout, start condition, or stop condition).

[Syntax]

```
void    r_<Config_RIIC0>_error_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_receive_interrupt

This API function executes processing in response to receive data full interrupts.

[Syntax]

```
void        r_<Config_RIIC0>_receive_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_transmit_interrupt

This API function executes processing in response to transmit data empty interrupts.

[Syntax]

```
void        r_<Config_RIIC0>_transmit_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_transmitend_interrupt

This API function executes processing in response to transmit end interrupts.

[Syntax]

```
void        r_<Config_RIIC0>_transmitend_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_callback_error

This API function executes processing specific to the detection of a loss in arbitration, NACK, timeout or communication sequence error among the sources of communication error interrupts or communication event generation interrupts.

Remark This API function is called from r_<Config_RIIC0>_error_interrupt as a callback routine.

[Syntax]

```
void r_<Config_RIIC0>_callback_error ( MD_STATUS status );
```

[Argument(s)]

I/O	Argument	Description
I	MD_STATUS *status*;	Interrupt sources MD_ERROR1: Detection of loss in arbitration MD_ERROR2: Detection of timeout MD_ERROR3: Detection of NACK MD_ERROR4: Detection of communication sequence error

[Return value]

None.

r_<Config_RIIC0>_callback_transmitend

This API function executes processing specific to the detection of a stop condition in slave transmission among the sources of communication error interrupts or communication event generation interrupts.

Remark1. This API function is called from r_<Config_RIIC0>_error_interrupt as a callback routine.

Remark2. To execute slave transmission, call R_<Config_RIIC0>_Slave_Send.

[Syntax]

```
void        r_<Config_RIIC0>_callback_transmitend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RIIC0>_callback_receiveend

This API function executes processing specific to the detection of a stop condition in slave reception among the sources of communication error interrupts or communication event generation interrupts.

Remark1. This API function is called from r_<Config_RIIC0>_error_interrupt as a callback routine.

Remark2. To execute slave reception, call R_<Config_RIIC0>_Slave_Receive.

[Syntax]

```
void        r_<Config_RIIC0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

4.2.19.1 Usage example

Repeating slave reception four times:

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t g_riic2_rx_buf[2];
void main(void)
{
    /* Start the RIIC2 Bus Interface */
    R_Config_RIIC2_Start();

    /* Read data from a master device */
    R_Config_RIIC2_Slave_Receive((uint8_t *)g_riic2_rx_buf, 2U);

    while (1U)
    {
        nop();
    }
}
```

Config_RIIC2_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_riic2_rx_buf[2];
volatile uint8_t g_riic2_rx_cnt;
/* End user code. Do not edit comment generated here */

void R_Config_RIIC2_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_riic2_rx_cnt = 0U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_RIIC2_callback_receiveend(void)
{
    /* Start user code for r_Config_RIIC2_callback_receiveend. Do not edit comment generated
here */
    if ((++g_riic2_rx_cnt) < 4U)
    {
        /* Read data from a master device */
        R_Config_RIIC2_Slave_Receive((uint8_t *)g_riic2_rx_buf, 2U);
    }
    else
    {
        /* Stop the RIIC2 Bus Interface */
        R_Config_RIIC2_Stop();
    }
    /* End user code. Do not edit comment generated here */
}
```

4.2.20 Interrupt Controller

The Code Generator outputs the following API functions for the interrupt controller.

Table4.21 API Functions: [Interrupt Controller]

API Function Name	Function
R_<Config_ICU>_Create	Executes initialization processing that is required before controlling the interrupt controller.
R_<Config_ICU>_IRQn_Start	Enables the detection of an external pin interrupt.
R_<Config_ICU>_IRQn_Stop	Disables the detection of an external pin interrupt.
R_<Config_ICU>_Software_Start	Enables the detection of a software interrupt.
R_<Config_ICU>_Software_Stop	Disables the detection of a software interrupt.
R_<Config_ICU>_SoftwareInterrupt_Generate	Generates a software interrupt.
R_<Config_ICU>_Software2_Start	Enables the detection of software interrupt 2.
R_<Config_ICU>_Software2_Stop	Disables the detection of software interrupt 2.
R_<Config_ICU>_SoftwareInterrupt2_Generate	Generates software interrupt 2.
R_<Config_ICU>_Create_UserInit	Executes user-specific initialization processing for the interrupt controller.
r_<Config_ICU>_irqn_interrupt	Executes processing in response to external pin interrupts.
r_<Config_ICU>_software_interrupt	Executes processing in response to software interrupts.
r_<Config_ICU>_software2_interrupt	Executes processing in response to software interrupts 2.
r_<Config_ICU>_nmi_interrupt	Executes processing in response to NMI pin interrupts.

R_<Config_ICU>_Create

This API function executes initialization processing that is required before controlling the interrupt controller.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_ICU>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_IRQ*n*_Start

This API function enables the detection of an external pin interrupt.

[Syntax]

```
void        R_<Config_ICU>_IRQn_Start ( void );
```

Remark *n* is the number of an IRQ pin.

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_IRQ*n*_Stop

This API function disables the detection of an external pin interrupt.

[Syntax]

```
void        R_<Config_ICU>_IRQn_Stop ( void );
```

Remark *n* is the number of an IRQ pin.

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_Software_Start

This API function enables the detection of a software interrupt.

[Syntax]

```
void        R_<Config_ICU>_Software_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_Software_Stop

This API function disables the detection of a software interrupt.

[Syntax]

```
void        R_<Config_ICU>_Software_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_SoftwareInterrupt_Generate

This API function generates a software interrupt.

Remark r_<Config_ICU>_software_interrupt is called in response to this API function.

[Syntax]

```
void R_<Config_ICU>_SoftwareInterrupt_Generate ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_Software2_Start

This API function enables the detection of software interrupt 2.

[Syntax]

```
void        R_<Config_ICU>_Software2_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_Software2_Stop

This API function disables the detection of software interrupt 2.

[Syntax]

```
void        R_<Config_ICU>_Software2_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_SoftwareInterrupt2_Generate

This API function generates software interrupt 2.

Remark r_<Config_ICU>_software2_interrupt is called in response to this API function.

[Syntax]

```
void R_<Config_ICU>_SoftwareInterrupt2_Generate ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_ICU>_Create_UserInit

This API function executes user-specific initialization processing for the interrupt controller.

Remark This API function is called from R_<Config_ICU>_Create as a callback routine.

[Syntax]

```
void R_<Config_ICU>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_ICU>_irq*n*_interrupt

This API function executes processing in response to external pin interrupts.

[Syntax]

```
void        r_<Config_ICU>_irqn_interrupt ( void );
```

Remark *n* is the number of an IRQ pin.

[Argument(s)]

None.

[Return value]

None.

r_<Config_ICU>_software_interrupt

This API function executes processing in response to software interrupts.

Remark This API function is called as the interrupt handler for software interrupts, which are issued by calling R_<Config_ICU>_SoftwareInterrupt_Generate.

[Syntax]

```
void r_<Config_ICU>_software_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_ICU>_software2_interrupt

This API function executes processing in response to software interrupts 2.

Remark This API function is called as the interrupt handler for software interrupts 2, which are issued by calling R_<Config_ICU>_SoftwareInterrupt2_Generate.

[Syntax]

```
void r_<Config_ICU>_software2_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_ICU>_nmi_interrupt

This API function executes processing in response to NMI pin interrupts.

[Syntax]

```
void        r_<Config_ICU>_nmi_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Releasing the MCU from the sleep mode in response to the detection of an external interrupt:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Enable IRQ0 interrupt */
    R_Config_ICU_IRQ0_Start();

    /* Enable sleep mode */
    R_Config_LPC_Sleep();

    while (1U)
    {
        nop();
    }
}
```

Config_ICU_user.c

```
/* Start user code for include. Do not edit comment generated here */
#include "r_smc_entry.h"
/* End user code. Do not edit comment generated here */

static void r_Config_ICU_irq0_interrupt(void)
{
    /* Start user code for r_Config_ICU_irq0_interrupt. Do not edit comment generated here */
    /* Allow sleep mode return clock to be changed */
    R_Config_LPC_ChangeSleepModeReturnClock(RETURN_MAIN_CLOCK);
    /* End user code. Do not edit comment generated here */
}
```

4.2.21 Low Power Consumption

The Code Generator outputs the following API functions for the low power consumption.

Table4.22 API Functions: [Low Power Consumption]

API Function Name	Function
R_<Config_LPC>_Create	Executes initialization processing that is required before controlling the low power consumption.
R_<Config_LPC>_AllModuleClockStop	Stops supply of the clock signal to all modules.
R_<Config_LPC>_Sleep	Places the MCU in sleep mode as a low-power state.
R_<Config_LPC>_DeepSleep	Places the MCU in deep sleep mode as a low-power state.
R_<Config_LPC>_SoftwareStandby	Places the MCU in software standby mode as a low-power state.
R_<Config_LPC>_DeepSoftwareStandby	Places the MCU in deep software standby mode as a low-power state.
R_<Config_LPC>_ChangeOperatingPowerControl	Changes the operating power control mode of the MCU.
R_<Config_LPC>_ChangeSleepModeReturnClock	Sets the clock source to be selected upon release from sleep mode.
R_<Config_LPC>_Create_UserInit	Executes user-specific initialization processing for the low power consumption.
R_<Config_LPC>_SetVOLSR_PGAVLS	Sets the programmable gain amplifier

R_<Config_LPC>_Create

This API function executes initialization processing that is required before controlling the low power consumption.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_LPC>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LPC>_AllModuleClockStop

This API function stops supply of the clock signal to all modules.

[Syntax]

```
MD_STATUS    R_<Config_LPC>_AllModuleClockStop ( void );
```

[Argument(s)]

None.

[Return value]

Macro	Description
MD_OK	Normal end

R_<Config_LPC>_Sleep

This API function places the MCU in sleep mode as a low-power state.

[Syntax]

```
MD_STATUS    R_<Config_LPC>_Sleep ( void );
```

[Argument(s)]

None.

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR1	Abnormal end

R_<Config_LPC>_DeepSleep

This API function places the MCU in deep sleep mode as a low-power state.

[Syntax]

```
MD_STATUS    R_<Config_LPC>_DeepSleep ( void );
```

[Argument(s)]

None.

[Return value]

Macro	Description
MD_OK	Normal end

R_<Config_LPC>_SoftwareStandby

This API function places the MCU in software standby mode as a low-power state.

[Syntax]

```
MD_STATUS    R_<Config_LPC>_SoftwareStandby ( void );
```

[Argument(s)]

None.

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR1	Abnormal end

R_<Config_LPC>_DeepSoftwareStandby

This API function places the MCU in deep software standby mode as a low-power state.

[Syntax]

```
MD_STATUS    R_<Config_LPC>_DeepSoftwareStandby ( void );
```

[Argument(s)]

None.

[Return value]

Macro	Description
MD_OK	Normal end

R_<Config_LPC>_ChangeOperatingPowerControl

This API function changes the operating power control mode of the MCU.

[Syntax]

```
MD_STATUS      R_<Config_LPC>_ChangeOperatingPowerControl ( operating_mode_t mode );
```

[Argument(s)]

For RX130 or RX230/RX231

I/O	Argument	Description
I	operating_mode_t *mode*;	Operating power control mode type HIGH_SPEED: High-speed operating mode MIDDLE_SPEED: Medium-speed operating mode LOW_SPEED: Low-speed operating mode

For other devices

I/O	Argument	Description
I	operating_mode_t *mode*;	Operating power control mode type HIGH_SPEED: High-speed operating mode LOW_SPEED1: Low-speed operating mode 1 LOW_SPEED2: Low-speed operating mode 2

[Return value]

For RX130 or RX230/RX231

Macro	Description
MD_OK	Normal end
MD_ERROR1	The transition to the low-speed operating mode did not end normally.
MD_ARGERROR	The specification of argument *mode* is invalid.

For other devices

Macro	Description
MD_OK	Normal end
MD_ERROR1	The transition to the low-speed operating mode 1 did not end normally.
MD_ERROR2	The transition to the low-speed operating mode 2 did not end normally.
MD_ARGERROR	The specification of argument *mode* is invalid.

R_<Config_LPC>_ChangeSleepModeReturnClock

This API function sets the clock source to be selected upon release from sleep mode.

[Syntax]

MD_STATUS	R_<Config_LPC>_ChangeSleepModeReturnClock (return_clock_t *clock*);

[Argument(s)]

For RX130 or RX230/RX231

I/O	Argument	Description
I	return_clock_t *clock*;	Clock source type RETURN_LOCO: Low-speed on-chip oscillator RETURN_HOCO: High-speed on-chip oscillator RETURN_MAIN_CLOCK: Main clock oscillator RETURN_DISABLE: The clock source is not switched.

For other devices

I/O	Argument	Description
I	return_clock_t *clock*;	Clock source type RETURN_HOCO: High-speed on-chip oscillator RETURN_MAIN_CLOCK: Main clock oscillator RETURN_DISABLE: The clock source is not switched.

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR1	The specification of the clock source that is currently set is invalid.
MD_ARGERROR	The specification of argument *clock* is invalid.

R_<Config_LPC>_Create_UserInit

This API function executes user-specific initialization processing for the low power consumption.

Remark This API function is called from R_<Config_LPC>_Create as a callback routine.

[Syntax]

```
void R_<Config_LPC>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LPC>_SetVOLSR_PGAVLS

This API function sets the PGA operating condition setting (VOLSR.PGAVLS bit).

[Syntax]

```
MD_STATUS        R_<Config_LPC>_SetVOLSR_PGAVLS ( uint8_t pgavls_bit );
```

[Argument(s)]

I/O	Argument	Description
I	uint8_t pgavls_bit;	PGAVLS bit value 0u :The AVCC voltage is at least 4.0 V, and the pseudo-differential input of the PGAs are enabled and negative voltages is to be input to the pins. 1u :The AVCC voltage is lower than 4.0 V, or negative voltages are not to be input to the pins.

None.

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR1	Abnormal end

Usage example

Refer to “Example” in Interrupt Controller.

4.2.22 Low Power Timer

The Code Generator outputs the following API functions for the low-power timer.

Table4.23 API Functions: [Low Power Timer]

API Function Name	Function
R_<Config_LPT>_Create	Executes initialization processing that is required before controlling the low-power timer.
R_<Config_LPT>_Start	Starts counting by the counter.
R_<Config_LPT>_Stop	Stops counting by the counter.
R_<Config_LPT>_Create_UserInit	Executes user-specific initialization processing for the low-power timer.

R_<Config_LPT>_Create

This API function executes initialization processing that is required before controlling the low-power timer.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_LPT>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LPT>_Start

This API function starts counting by the counter.

[Syntax]

```
void        R_<Config_LPT>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LPT>_Stop

This API function stops counting by the counter.

[Syntax]

```
void        R_<Config_LPT>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LPT>_Create_UserInit

This API function executes user-specific initialization processing for the low-power timer.

Remark This API function is called from R_<Config_LPT>_Create as a callback routine.

[Syntax]

```
void R_<Config_LPT>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

None.

4.2.23 Normal Mode Timer

The Code Generator outputs the following API functions for normal mode timer (MTU or TPU).

Table4.24 API Functions: [Normal Mode Timer]

API Function Name	Function
R_<Config_MTU0>_Create	Executes initialization processing that is required before controlling the normal mode timer (MTU or TPU).
R_<Config_MTU0>_Start	Starts counting by the counter.
R_<Config_MTU0>_Stop	Stops counting by the counter.
R_<Config_MTU0>_Create_UserInit	Executes user-specific initialization processing for the normal mode timer (MTU or TPU).
r_<Config_MTU0>_tgimn_interrupt	Executes processing in response to input capture or compare match interrupts. (The name of this API function varies with the resource.)
r_<Config_MTU0>_tginm_interrupt	
r_<Config_MTU0>_tcivn_interrupt	Executes processing in response to overflow interrupts. (The name of this API function varies with the resource.)
r_<Config_MTU0>_tcinv_interrupt	

R_<Config_MTU0>_Create

This API function executes initialization processing that is required before controlling a timer (MTU or TPU) in normal mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_MTU0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU0>_Start

This API function starts counting by the counter.

[Syntax]

```
void        R_<Config_MTU0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU0>_Stop

This API function stops counting by the counter.

[Syntax]

```
void        R_<Config_MTU0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU0>_Create_UserInit

This API function executes user-specific initialization processing for a timer (MTU or TPU) in normal mode.

Remark This API function is called from R_<Config_MTU0>_Create as a callback routine.

[Syntax]

```
void R_<Config_MTU0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU0>_tgi*mn*_interrupt

r_<Config_MTU0>_tgi*nm*_interrupt

This API function executes processing in response to input capture or compare match interrupts.

[Syntax]

```
void    r_<Config_MTU0>_tgimn_interrupt ( void );
```

```
void    r_<Config_MTU0>_tginm_interrupt ( void );
```

Remark1. *n* is a channel number and *m* is the number of a timer general register.

Remark2. The name of this API function varies with the resource.

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU0>_tciv*n*_interrupt

r_<Config_MTU0>_tci*n*v_interrupt

This API function executes processing in response to overflow interrupts.

[Syntax]

void	r_<Config_MTU0>_tciv*n*_interrupt (void);

void	r_<Config_MTU0>_tci*n*v_interrupt (void);

Remark1. *n* is a channel number.

Remark2. The name of this API function varies with the resource.

[Argument(s)]

None.

[Return value]

None.

Usage example

With the timer operating in a one-shot manner:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start MTU channel 0 counter */
    R_Config_MTU0_Start();

    while (1U)
    {
        nop();
    }
}
```

<Config_MTU0_user.c

```
static void r_<Config_MTU0_tgia0_interrupt(void)
{
    /* Start user code for r_<Config_MTU0_tgia0_interrupt. Do not edit comment generated here */
    /* Stop MTU channel 0 counter */
    R_Config_MTU0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.24 Phase Counting Mode Timer

The Code Generator outputs the following API functions for phase counting mode timers (MTU or TPU).

Table4.25 API Functions: [Phase Counting Mode Timer]

API Function Name	Function
R_<Config_MTU1>_Create	Executes initialization processing that is required before controlling the phase counting mode timer (MTU or TPU).
R_<Config_MTU1>_Start	Starts counting by the counter.
R_<Config_MTU1>_Stop	Stops counting by the counter.
R_<Config_MTU1_MTU2>_MTU_Start	Starts all MTU channels' counters simutenously for 32-bit cascading operation
R_<Config_MTU1_MTU2>_MTU_Stop	Stops all MTU channels' counters simutenously for 32-bit cascading operation
R_<Config_MTU1>_Create_UserInit	Executes user-specific initialization processing for the phase counting mode timer (MTU or TPU).
r_<Config_MTU1>_tgimn_interrupt	Executes processing in response to input capture or compare match interrupts. (The name of this API function varies with the resource.)
r_<Config_MTU1>_tginm_interrupt	
r_<Config_MTU1>_tcivn_interrupt	Executes processing in response to overflow interrupts. (The name of this API function varies with the resource.)
r_<Config_MTU1>_tcinv_interrupt	
r_<Config_MTU1>_tciun_interrupt	Executes processing in response to underflow interrupts. (The name of this API function varies with the resource.)
r_<Config_MTU1>_tcinu_interrupt	

R_<Config_MTU1>_Create

This API function executes initialization processing that is required before controlling a timer (MTU or TPU) in phase counting mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_MTU1>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU1>_Start

This API function starts counting by the counter.

[Syntax]

```
void        R_<Config_MTU1>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU1>_Stop

This API function stops counting by the counter.

[Syntax]

```
void        R_<Config_MTU1>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU1_MTU2>_MTU_Start

This API function starts MTU channels' counter simutenously for 32-bit cascading operation, other channels related to the ones used for cascading operation are configurable via cascading mode GUI

[Syntax]

```
void        R_<Config_MTU1_MTU2>MTU_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU1_MTU2>_MTU_Stop

This API function stops MTU channels' counter simutenously for 32-bit cascading operation, other channels related to the ones used for cascading operation are configurable via cascading mode GUI

[Syntax]

```
void        R_<Config_MTU1_MTU2>MTU_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU1>_Create_UserInit

This API function executes user-specific initialization processing for a timer (MTU or TPU) in phase counting mode.

Remark This API function is called from R_<Config_MTU1>_Create as a callback routine.

[Syntax]

```
void R_<Config_MTU1>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU1>_tgi*mn*_interrupt

r_<Config_MTU1>_tgi*nm*_interrupt

This API function executes processing in response to input capture or compare match interrupts.

[Syntax]

```
void    r_<Config_MTU1>_tgimn_interrupt ( void );
```

```
void    r_<Config_MTU1>_tginm_interrupt ( void );
```

Remark1. *n* is a channel number and *m* is the number of a timer general register.

Remark2. The name of this API function varies with the resource.

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU1>_tciv*n*_interrupt

r_<Config_MTU1>_tci*n*v_interrupt

This API function executes processing in response to overflow interrupts.

[Syntax]

void	r_<Config_MTU1>_tciv*n*_interrupt (void);

void	r_<Config_MTU1>_tci*n*v_interrupt (void);

Remark1. *n* is a channel number.

Remark2. The name of this API function varies with the resource.

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU1>_tciu*n*_interrupt

r_<Config_MTU1>_tci*n*u_interrupt

This API function executes processing in response to underflow interrupts.

[Syntax]

void	r_<Config_MTU1>_tciu*n*_interrupt (void);

void	r_<Config_MTU1>_tci*n*u_interrupt (void);

Remark1. *n* is a channel number.

Remark2. The name of this API function varies with the resource.

[Argument(s)]

None.

[Return value]

None.

Usage example

Determining the direction of rotation from the 2-phase pulses input from a rotary encoder:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the MTU1 channel counter */
    R_Config_MTU1_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_MTU1_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_mtu1_dir;
/* End user code. Do not edit comment generated here */

void R_Config_MTU1_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    /* Clear state */
    g_mtu1_dir = 0U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_MTU1_tgia1_interrupt(void)
{
    /* Start user code for r_Config_MTU1_tgia1_interrupt. Do not edit comment generated here */
    /* Set CW state */
    g_mtu1_dir = 1U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_MTU1_tgib1_interrupt(void)
{
    /* Start user code for r_Config_MTU1_tgib1_interrupt. Do not edit comment generated here */
    /* Set CCW state */
    g_mtu1_dir = 2U;
    /* End user code. Do not edit comment generated here */
}
```

4.2.25 Port Output Enable

The Code Generator outputs the following API functions for the port output enable module.

Table4.26 API Functions: [Port Output Enable]

API Function Name	Function
R_<Config_POE>_Create	Executes initialization processing that is required before controlling the port output enable module.
R_<Config_POE>_Start	Enables output enable interrupts.
R_<Config_POE>_Stop	Disables output enable interrupts.
R_<Config_POE>_Set_HiZ_MTUn	Places the pins of the MTU*n* in the high-impedance state.
R_<Config_POE>_Clear_HiZ_MTUn	Releases the pins of the MTU*n* from the high-impedance state.
R_<Config_POE>_Set_HiZ_GPTn	Places the pins of the GPT*n* in the high-impedance state.
R_<Config_POE>_Clear_HiZ_GPTn	Releases the pins of the GPT*n* from the high-impedance state.
R_<Config_POE>_Create_UserInit	Executes user-specific initialization processing for the port output enable module.
r_<Config_POE>_oein_interrupt	Executes processing in response to output enable interrupts.

R_<Config_POE>_Create

This API function executes initialization processing that is required before controlling the port output enable module.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_<Config_POE>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_POE>_Start

This API function enables output enable interrupts.

Remark This API function is empty, if interrupt is not configured for POE.

[Syntax]

```
void    R_<Config_POE>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_POE>_Stop

This API function disables output enable interrupts.

Remark This API function is empty, if interrupt is not configured for POE.

[Syntax]

```
void    R_<Config_POE>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_POE>_Set_HiZ_MTU*n*

This API function places the pins of the MTU*n* in the high-impedance state.

[Syntax]

void	R_<Config_POE>_Set_HiZ_MTU*n* (void);

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

R_<Config_POE>_Clear_HiZ_MTU*n*

This API function releases the pins of the MTU*n* from the high-impedance state.

[Syntax]

void	R_<Config_POE>_Clear_HiZ_MTU*n* (void);

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

R_<Config_POE>_Set_HiZ_GPT*n*

This API function places the pins of the GPT*n* in the high-impedance state.

[Syntax]

void	R_<Config_POE>_Set_HiZ_GPT*n* (void);

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

R_<Config_POE>_Clear_HiZ_GPT*n*

This API function releases the pins of the GPT*n* from the high-impedance state.

[Syntax]

void	R_<Config_POE>_Clear_HiZ_GPT*n* (void);

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

R_<Config_POE>_Create_UserInit

This API function executes user-specific initialization processing for the port output enable module.

Remark This API function is called from R_<Config_POE>_Create as a callback routine.

[Syntax]

```
void R_<Config_POE>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_POE>_oei*n*_interrupt

This API function executes processing in response to output enable interrupts.

[Syntax]

```
void        r_<Config_POE>_oei n_interrupt ( void );
```

Remark *n* is the number of an output enable interrupt source.

[Argument(s)]

None.

[Return value]

None.

Usage example

Placing the pins of the MTU0 in the high-impedance state in response to an output enable interrupt:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the POE module */
    R_Config_POE_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_POE_user.c

```
void r_Config_POE_oei1_interrupt(void)
{
    /* Start user code for r_Config_POE_oei1_interrupt. Do not edit comment generated here */
    /* Stop the POE module */
    R_Config_POE_Stop();

    /* Place MTU0 pins in high-impedance */
    R_Config_POE_Set_HiZ_MTU0();
    /* End user code. Do not edit comment generated here */
}
```

4.2.26 I/O Port

The Code Generator outputs the following API functions for the I/O ports.

Table4.27 API Functions: [I/O Port]

API Function Name	Function
R_<Config_PORT>_Create	Executes initialization processing that is required before controlling the I/O ports.
R_<Config_PORT>_Create_UserInit	Executes user-specific initialization processing for the I/O ports.

R_<Config_PORT>_Create

This API function executes initialization processing that is required before controlling the I/O ports.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_PORT>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_PORT>_Create_UserInit

This API function executes user-specific initialization processing for the I/O ports.

Remark This API function is called from R_<Config_PORT>_Create as a callback routine.

[Syntax]

```
void    R_<Config_PORT>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

None.

4.2.27 Programmable Pulse Generator

The Code Generator outputs the following API functions for the programmable pulse generator.

Table4.28 API Functions: [Programmable Pulse Generator]

API Function Name	Function
R_<Config_PPG0>_Create	Executes initialization processing that is required before controlling the programmable pulse generator.
R_<Config_PPG0>_Create_UserInit	Executes user-specific initialization processing for the programmable pulse generator.

R_<Config_PPG0>_Create

This API function executes initialization processing that is required before controlling the programmable pulse generator.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_PPG0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_PPG0>_Create_UserInit

This API function executes user-specific initialization processing for the programmable pulse generator.

Remark This API function is called from R_<Config_PPG0>_Create as a callback routine.

[Syntax]

```
void R_<Config_PPG0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

None.

4.2.28 PWM Mode Timer

The Code Generator outputs the following API functions for timers (MTU or TPU) in PWM mode.

Table4.29 API Functions: [PWM Mode Timer]

API Function Name	Function
R_<Config_MTU0>_Create	Executes initialization processing that is required before controlling the PWM mode timer (MTU or TPU).
R_<Config_MTU0>_Start	Starts counting by the counter.
R_<Config_MTU0>_Stop	Stops counting by the counter.
R_<Config_MTU0>_Create_UserInit	Executes user-specific initialization processing for the PWM mode timer (MTU or TPU).
r_<Config_MTU0>_tgimn_interrupt	Executes processing in response to compare match interrupts. (The name of this API function varies with the resource.)
r_<Config_MTU0>_tginm_interrupt	
r_<Config_MTU0>_tcivn_interrupt	Executes processing in response to overflow interrupts. (The name of this API function varies with the resource.)
r_<Config_MTU0>_tcinv_interrupt	

R_<Config_MTU0>_Create

This API function executes initialization processing that is required before controlling a timer (MTU or TPU) in PWM mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_MTU0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU0>_Start

This API function starts counting by the counter.

[Syntax]

```
void        R_<Config_MTU0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU0>_Stop

This API function stops counting by the counter.

[Syntax]

```
void        R_<Config_MTU0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU0>_Create_UserInit

This API function executes user-specific initialization processing for a timer (MTU or TPU) in PWM mode.

Remark This API function is called from R_<Config_MTU0>_Create as a callback routine.

[Syntax]

```
void R_<Config_MTU0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU0>_tgi*mn*_interrupt

r_<Config_MTU0>_tgi*nm*_interrupt

This API function executes processing in response to compare match interrupts.

[Syntax]

void	r_<Config_MTU0>_tgi*mn*_interrupt (void);

void	r_<Config_MTU0>_tgi*nm*_interrupt (void);

Remark1. *n* is a channel number and *m* is the number of a timer general register.

Remark2. The name of this API function varies with the resource.

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU0>_tciv*n*_interrupt

r_<Config_MTU0>_tci*n*v_interrupt

This API function executes processing in response to overflow interrupts.

[Syntax]

void	r_<Config_MTU0>_tciv*n*_interrupt (void);

void	r_<Config_MTU0>_tci*n*v_interrupt (void);

Remark1. *n* is a channel number.

Remark2. The name of this API function varies with the resource.

[Argument(s)]

None.

[Return value]

None.

Usage example

Output of ten PWM pulses:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start MTU channel 0 counter */
    R_Config_MTU0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_MTU0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_mtu0_cnt;
/* End user code. Do not edit comment generated here */
void R_Config_MTU0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    /* Reset the countor */
    g_mtu0_cnt = 0U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_MTU0_tgia0_interrupt(void)
{
    /* Start user code for r_Config_MTU0_tgia0_interrupt. Do not edit comment generated here */
    if ((++g_mtu0_cnt) > 9U)
    {
        /* Stop MTU channel 0 counter */
        R_Config_MTU0_Stop();
    }
    /* End user code. Do not edit comment generated here */
}
```

4.2.29 Realtime Clock

The Code Generator outputs the following API functions for the realtime clock.

Table4.30 API Functions: [Realtime Clock]

API Function Name	Function
R_<Config_RTC>_Create	Executes initialization processing that is required before controlling the realtime clock.
R_<Config_RTC>_Start	Starts counting by the counters.
R_<Config_RTC>_Stop	Stops counting by the counters.
R_<Config_RTC>_Restart	Stops the counters, initializes the counter values, and then restarts the counters in calendar count mode.
R_<Config_RTC>_Restart_BinaryCounter	Stops the counters, clears the counter values to 0, and then restarts the counters in binary count mode.
R_<Config_RTC>_Set_CalendarCounterValue	Specifies calendar values.
R_<Config_RTC>_Get_CalendarCounterValue	Gets the calendar values from the counter registers.
R_<Config_RTC>_Get_CalendarTimeCaptureValuen	Gets the calendar values from the capture registers.
R_<Config_RTC>_Set_BinaryCounterValue	Specifies a value for binary counting.
R_<Config_RTC>_Get_BinaryCounterValue	Gets the binary counter value from the counter registers.
R_<Config_RTC>_Get_BinaryTimeCaptureValuen	Gets the binary counter value from the capture registers.
R_<Config_RTC>_Set_RTCOUTOn	Specifies the frequency of output through the RTCOUT pin and starts the output.
R_<Config_RTC>_Set_RTCOUTOff	Stops output through the RTCOUT pin.
R_<Config_RTC>_Set_CalendarAlarm	Specifies the conditions for generating alarm interrupts and enables the detection of the interrupts. (Calendar count mode)
R_<Config_RTC>_Set_BinaryAlarm	Specifies the conditions for generating alarm interrupts and enables the detection of the interrupts. (Binary count mode)
R_<Config_RTC>_Set_ConstPeriodInterruptOn	Specifies the interval of periodic interrupts and enables the detection of the interrupts.
R_<Config_RTC>_Set_ConstPeriodInterruptOff	Disables the detection of periodic interrupts.
R_<Config_RTC>_Set_CarryInterruptOn	Enables the detection of carry interrupts.
R_<Config_RTC>_Set_CarryInterruptOff	Disables the detection of carry interrupts.
R_<Config_RTC>_Enable_Alarm_Interrupt	Enables the alarm interrupt.
R_<Config_RTC>_Disable_Alarm_Interrupt	Disables the alarm interrupt.
R_<Config_RTC>_Create_UserInit	Executes user-specific initialization processing for the realtime clock.
r_<Config_RTC>_alm_interrupt	Executes processing in response to alarm interrupts.
r_<Config_RTC>_prd_interrupt	Executes processing in response to periodic interrupts.
r_<Config_RTC>_cup_interrupt	Executes processing in response to carry interrupts.

R_<Config_RTC>_Create

This API function executes initialization processing that is required before controlling the realtime clock.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_RTC>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Start

This API function starts counting by the counters.

[Syntax]

```
void        R_<Config_RTC>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Stop

This API function stops counting by the counters.

[Syntax]

```
void        R_<Config_RTC>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Restart

This API function stops the counters, initializes the counter values, and then restarts the counters in calendar count mode.

[Syntax]

```
void    R_<Config_RTC>_Restart ( rtc_calendarcounter_value_t counter_write_val );
```

[Argument(s)]

I/O	Argument	Description
I	rtc_calendarcounter_value_t *counter_write_val*;	Initial values (year, month, date, day-of-week, hour, minute, and second)

Remark The following shows the structure that holds the initial value arguments, rtc_calendarcounter_value_t.

```
typedef struct {
    uint8_t rseccnt;        /*  Second  */
    uint8_t rmincnt;        /*  Minute  */
    uint8_t rhrcnt;         /*  Hour  */
    uint8_t rdaycnt;        /*  Date  */
    uint8_t rwkcnt;         /*  Day-of-week (0: Sunday; 6: Saturday)  */
    uint8_t rmoncnt;        /*  Month  */
    uint16_t ryrcnt;        /*  Year  */
} rtc_calendarcounter_value_t;
```

[Return value]

None.

R_<Config_RTC>_Restart_BinaryCounter

This API function stops the counters, clears the counter values to 0, and then restarts the counters in binary count mode.

[Syntax]

```
void        R_<Config_RTC>_Restart_BinaryCounter ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Set_CalendarCounterValue

This API function specifies calendar values.

[Syntax]

```
void    R_<Config_RTC>_Set_CalendarCounterValue ( rtc_calendarcounter_value_t counter_write_val );
```

[Argument(s)]

I/O	Argument	Description
I	rtc_calendarcounter_value_t *counter_write_val*;	Calendar values (year, month, date, day-of-week, hour, minute, and second)

Remark The following shows the structure that holds the calendar value arguments, rtc_calendarcounter_value_t.

```
typedef struct {
    uint8_t rseccnt;        /*  Second  */
    uint8_t rmincnt;        /*  Minute  */
    uint8_t rhrcnt;         /*  Hour  */
    uint8_t rdaycnt;        /*  Date  */
    uint8_t rwkcnt;         /*  Day-of-week (0: Sunday; 6: Saturday)  */
    uint8_t rmoncnt;        /*  Month  */
    uint16_t ryrcnt;        /*  Year  */
} rtc_calendarcounter_value_t;
```

[Return value]

None.

R_<Config_RTC>_Get_CalendarCounterValue

This API function gets the calendar values from the counter registers.

[Syntax]

```
void R_<Config_RTC>_Get_CalendarCounterValue ( rtc_calendarcounter_value_t * const counter_read_val );
```

[Argument(s)]

I/O	Argument	Description
O	rtc_calendarcounter_value_t * const *counter_read_val*;	Pointer to the location where the acquired calendar values (year, month, date, day-of-week, hour, minute, and second) are to be stored

Remark The following shows the structure that holds the calendar value arguments, rtc_calendarcounter_value_t.

```
typedef struct {
    uint8_t rseccnt;    /*  Second  */
    uint8_t rmincnt;    /*  Minute  */
    uint8_t rhrcnt;     /*  Hour  */
    uint8_t rdaycnt;    /*  Date  */
    uint8_t rwkcnt;     /*  Day-of-week (0: Sunday; 6: Saturday)  */
    uint8_t rmoncnt;    /*  Month  */
    uint16_t ryrcnt;    /*  Year  */
} rtc_calendarcounter_value_t;
```

[Return value]

None.

R_<Config_RTC>_Get_CalendarTimeCaptureValue*n*

This API function gets the calendar values from the capture registers.

[Syntax]

```
void    R_<Config_RTC>_Get_CalendarTimeCaptureValuen ( rtc_calendarcounter_value_t * const
counter_read_val );
```

Remark *n* is a channel number.

[Argument(s)]

I/O	Argument	Description
O	rtc_calendarcounter_value_t * const *counter_read_val*;	Pointer to the location where the acquired calendar values (year, month, date, day-of-week, hour, minute, and second) are to be stored

Remark The following shows the structure that holds the calendar value arguments, rtc_calendarcounter_value_t.

```
typedef struct {
    uint8_t rseccnt;        /*  Second  */
    uint8_t rmincnt;        /*  Minute  */
    uint8_t rhrcnt;         /*  Hour  */
    uint8_t rdaycnt;        /*  Date  */
    uint8_t rwkcnt;         /*  Day-of-week (0: Sunday; 6: Saturday)  */
    uint8_t rmoncnt;        /*  Month  */
    uint16_t ryrcnt;        /*  Year  */
} rtc_calendarcounter_value_t;
```

[Return value]

None.

R_<Config_RTC>_Set_BinaryCounterValue

This API function specifies a value for binary counting.

[Syntax]

```
void    R_<Config_RTC>_Set_BinaryCounterValue ( uint32_t counter_write_val );
```

[Argument(s)]

I/O	Argument	Description
I	uint32_t *counter_write_val*;	Binary counter value

[Return value]

None.

R_<Config_RTC>_Get_BinaryCounterValue

This API function gets the binary counter value from the counter registers.

[Syntax]

```
void    R_<Config_RTC>_Get_BinaryCounterValue ( uint32_t * const counter_read_val );
```

[Argument(s)]

I/O	Argument	Description
O	uint32_t * const *counter_read_val*;	Pointer to the location where the acquired binary counter value is to be stored

[Return value]

None.

R_<Config_RTC>_Get_BinaryTimeCaptureValue*n*

This API function gets the binary counter value from the capture registers.

[Syntax]

```
void    R_<Config_RTC>_Get_BinaryTimeCaptureValuen ( uint32_t * const counter_read_val );
```

Remark *n* is a channel number.

[Argument(s)]

I/O	Argument	Description
O	uint32_t * const *counter_read_val*;	Pointer to the location where the acquired binary counter value is to be stored

[Return value]

None.

R_<Config_RTC>_Set_RTCOUTOn

This API function specifies the frequency of output through the RTCOUT pin and starts the output.

[Syntax]

```
void    R_<Config_RTC>_Set_RTCOUTOn ( rtc_rtcout_period_t rtcout_freq );
```

[Argument(s)]

I/O	Argument	Description
I	rtc_rtcout_period_t *rtcout_freq*;	Frequency of output through RTCOUT RTCOUT_1HZ: 1 Hz RTCOUT_64HZ: 64 Hz

[Return value]

None.

R_<Config_RTC>_Set_RTCOUTOff

This API function stops output through the RTCOUT pin.

[Syntax]

```
void        R_<Config_RTC>_Set_RTCOUTOff ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Set_CalendarAlarm

This API function specifies the conditions for generating alarm interrupts and enables the detection of the interrupts. (Calendar count mode)

[Syntax]

```
void    R_<Config_RTC>_Set_CalendarAlarm ( rtc_calendar_alarm_enable_t alarm_enable、
 rtc_calendar_alarm_value_t alarm_val );
```

[Argument(s)]

I/O	Argument	Description
I	rtc_calendar_alarm_enable_t *alarm_enable*;	Comparison flags (year, month, date, day-of-week, hour, minute, and second)
I	rtc_calendar_alarm_value_t *alarm_val*;	Calendar values (year, month, date, day-of-week, hour, minute, and second)

Remark1. The following shows the structure that holds the comparison flag arguments, rtc_calendar_alarm_enable_t.

```
typedef struct {
    uint8_t sec_enb;    /* Second (0x0: Not compared; 0x80: Compared) */
    uint8_t min_enb;    /* Minute (0x0: Not compared; 0x80: Compared) */
    uint8_t hr_enb;     /* Hour (0x0: Not compared; 0x80: Compared) */
    uint8_t day_enb;    /* Date (0x0: Not compared; 0x80: Compared) */
    uint8_t wk_enb;     /* Day-of-week (0x0: Not compared; 0x80: Compared) */
    uint8_t mon_enb;    /* Month (0x0: Not compared; 0x80: Compared) */
    uint8_t yr_enb;     /* Year (0x0: Not compared; 0x80: Compared) */
} rtc_calendar_alarm_enable_t;
```

Remark2. The following shows the structure that holds the calendar value arguments, rtc_calendar_alarm_value_t.

```
typedef struct {
    uint8_t rsecar;     /*  Second  */
    uint8_t rminar;     /*  Minute  */
    uint8_t rhrar;      /*  Hour  */
    uint8_t rdayar;     /*  Date  */
    uint8_t rwkar;      /*  Day-of-week (0: Sunday; 6: Saturday)  */
    uint8_t rmonar;     /*  Month  */
    uint16_t ryrar;     /*  Year  */
} rtc_calendar_alarm_value_t;
```

[Return value]

None.

R_<Config_RTC>_Set_BinaryAlarm

This API function specifies the conditions for generating alarm interrupts and enables the detection of the interrupts. (Binary count mode)

[Syntax]

void R_<Config_RTC>_Set_BinaryAlarm (uint32_t *alarm_enable*、 uint32_t *alarm_val*);

[Argument(s)]

I/O	Argument	Description
I	uint32_t *alarm_enable*;	Comparison flag 0x0: Not compared 0x1: Compared
I	uint32_t *alarm_val*;	Binary counter value

[Return value]

None.

R_<Config_RTC>_Set_ConstPeriodInterruptOn

This API function specifies the interval of periodic interrupts and enables the detection of the interrupts.

[Syntax]

```
void    R_<Config_RTC>_Set_ConstPeriodInterruptOn ( rtc_int_period_t period );
```

[Argument(s)]

I/O	Argument	Description
I	rtc_int_period_t *period*;	Interval of periodic interrupts PES_2_SEC: 2 seconds PES_1_SEC: 1 second PES_1_2_SEC: 1/2 second PES_1_4_SEC: 1/4 second PES_1_8_SEC: 1/8 second PES_1_16_SEC: 1/16 second PES_1_32_SEC: 1/32 second PES_1_64_SEC: 1/64 second PES_1_128_SEC: 1/128 second PES_1_256_SEC: 1/256 second

[Return value]

None.

R_<Config_RTC>_Set_ConstPeriodInterruptOff

This API function disables the detection of periodic interrupts.

[Syntax]

```
void        R_<Config_RTC>_Set_ConstPeriodInterruptOff ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Set_CarryInterruptOn

This API function enables the detection of carry interrupts.

[Syntax]

```
void        R_<Config_RTC>_Set_CarryInterruptOn ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Set_CarryInterruptOff

This API function disables the detection of carry interrupts.

[Syntax]

```
void        R_<Config_RTC>_Set_CarryInterruptOff ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Enable_Alarm_Interrupt

This API function enables the alarm interrupt request.

[Syntax]

```
void        R_<Config_RTC>_Enable_Alarm_Interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Disable_Alarm_Interrupt

This API function disables the alarm interrupt request.

[Syntax]

```
void        R_<Config_RTC>_Disable_Alarm_Interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RTC>_Create_UserInit

This API function executes user-specific initialization processing for the realtime clock.

Remark This API function is called from R_<Config_RTC>_Create as a callback routine.

[Syntax]

```
void R_<Config_RTC>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RTC>_alm_interrupt

This API function executes processing in response to alarm interrupts.

Remark This API function is called as the interrupt handler for alarm interrupts, which occur when the conditions specified in R_<Config_RTC>_Set_CalendarAlarm are satisfied.

[Syntax]

```
void    r_<Config_RTC>_alm_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RTC>_prd_interrupt

This API function executes processing in response to periodic interrupts.

Remark This API function is called as the interrupt handler for periodic interrupts, which occur when the period specified by the parameter period in R_<Config_RTC>_Set_ConstPeriodInterruptOn has elapsed.

[Syntax]

```
void        r_<Config_RTC>_prd_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RTC>_cup_interrupt

This API function executes processing in response to carry interrupts.

Remark This API function is called as the interrupt handler for carry interrupts, which occur in response to carrying to the second counter (RSECCNT) or binary counter 0 (BCNT0), or to carrying to the 64-Hz counter (R64CNT) while the counter (R64CNT) is being read.

[Syntax]

```
void        r_<Config_RTC>_cup_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Using alarm interrupts to implement virtual processing for leap second correction (turning back the clock from 23:59:59 to 23:59:58 on a scheduled day):

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start RTC counter */
    R_Config_RTC_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_RTC_user.c

```
/* Start user code for include. Do not edit comment generated here */
volatile rtc_calendarcounter_value_t counter_val;
/* End user code. Do not edit comment generated here */
static void r_Config_RTC_alm_interrupt(void)
{
    /* Start user code for r_Config_RTC_alm_interrupt. Do not edit comment generated here */
    /* Disable ALM interrupt */
    IEN(RTC、ALM) = 0U;

    /* Get RTC calendar counter value */
    R_Config_RTC_Get_CalendarCounterValue((rtc_calendarcounter_value_t *)&counter_val);

    /* Change the seconds */
    counter_val.rseccnt = 0x58U;

    /* Set RTC calendar counter value */
    R_Config_RTC_Set_CalendarCounterValue(counter_val);
    /* End user code. Do not edit comment generated here */
}
```

4.2.30 Remote Control Signal Receiver

The Code Generator outputs the following API functions for the remote control signal receiver.

Table4.31 API Functions: [Remote Control Signal Receiver]

API Function Name	Function
R_<Config_REMC0>_Create	Executes initialization processing that is required before controlling the remote control signal receiver.
R_<Config_REMC0>_Start	Starts operation of the remote control signal receiver.
R_<Config_REMC0>_Stop	Stops operation of the remote control signal receiver.
R_<Config_REMC0>_Read	Specifies the location where the received data are to be stored and the number of bytes to be received.
R_<Config_REMC0>_Create_UserInit	Executes user-specific initialization processing for the remote control signal receiver.
r_<Config_REMC0>_remcin_interrupt	Executes processing in response to REMC interrupts.
r_<Config_REMC0>_callback_comparematch	Executes processing in response to compare match interrupts.
r_<Config_REMC0>_callback_receiveerror	Executes processing in response to receive error interrupts.
r_<Config_REMC0>_callback_receiveend	Executes processing in response to data reception complete interrupts.
r_<Config_REMC0>_callback_bufferfull	Executes processing in response to receive buffer full interrupts.
r_<Config_REMC0>_callback_header	Executes processing in response to header pattern match interrupts.
r_<Config_REMC0>_callback_data0	Executes processing in response to data "0" pattern match interrupts.
r_<Config_REMC0>_callback_data1	Executes processing in response to data "1" pattern match interrupts.
r_<Config_REMC0>_callback_specialdata	Executes processing in response to special data pattern match interrupts.

R_<Config_REMC0>_Create

This API function executes initialization processing that is required before controlling the remote control signal receiver.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_REMC0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_REMC0>_Start

This API function starts operation of the remote control signal receiver.

[Syntax]

```
void        R_<Config_REMC0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_REMC0>_Stop

This API function stops operation of the remote control signal receiver.

[Syntax]

```
void        R_<Config_REMC0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_REMC0>_Read

This API function specifies the location where the received data are to be stored and the number of bytes to be received.

Remark This API function specifies the location where the received data read by the REMC interrupt routine at the end of data reception are to be stored.

[Syntax]

```
MD_STATUS    R_<Config_REMC0>_Read ( uint8_t * const rx_buf, uint8_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
O	uint8_t * const *rx_buf*	Pointer to the buffer where the received data are to be stored
I	uint8_t *rx_num*	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ERROR1	The specification of argument *rx_num* is invalid.

R_<Config_REMC0>_Create_UserInit

This API function executes user-specific initialization processing for the remote control signal receiver.

Remark This API function is called from R_<Config_REMC0>_Create as a callback routine.

[Syntax]

```
void R_<Config_REMC0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_remci*n*_interrupt

This API function executes processing in response to REMC interrupts.

[Syntax]

```
void        r_<Config_REMC0>_remci n_interrupt ( void );
```

Remark *n* is a channel number.

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_callback_comparematch

This API function executes processing in response to compare match interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```
void r_<Config_REMC0>_callback_comparematch ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_callback_receiveerror

This API function executes processing in response to receive error interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```
void r_<Config_REMC0>_callback_receiveerror ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_callback_receiveend

This API function executes processing in response to data reception complete interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```
void r_<Config_REMC0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_callback_bufferfull

This API function executes processing in response to receive buffer full interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```
void r_<Config_REMC0>_callback_bufferfull ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_callback_header

This API function executes processing in response to header pattern match interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```
void r_<Config_REMC0>_callback_header ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_callback_data0

This API function executes processing in response to data "0" pattern match interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```
void r_<Config_REMC0>_callback_data0 ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_callback_data1

This API function executes processing in response to data "1" pattern match interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```
void r_<Config_REMC0>_callback_data1 ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_REMC0>_callback_specialdata

This API function executes processing in response to special data pattern match interrupts.

Remark This API function is called as a callback routine from r_<Config_REMC0>_remcin_interrupt, which is the interrupt handler for REMC interrupts.

[Syntax]

```
void r_<Config_REMC0>_callback_specialdata ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Stopping operation of the remote control signal receiver at the end of data reception:

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t g_remc0_rx_buf[8];
void main(void)
{
    /* Start the REMC0 operation */
    R_Config_REMC0_Start();

    /* Read data from receive data buffer */
    R_Config_REMC0_Read((uint8_t *)g_remc0_rx_buf, 8U);

    while (1U)
    {
        nop();
    }
}
```

Config_REMC0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_remc0_rx_buf[8];
/* End user code. Do not edit comment generated here */
static void r_Config_REMC0_callback_receiveend(void)
{
    /* Start user code for r_Config_REMC0_callback_receiveend. Do not edit comment generated
here */
    /* Stop the REMC0 operation */
    R_Config_REMC0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.31 SCI/SCIF Asynchronous Mode

The Code Generator outputs the following API functions for the SCI/SCIF asynchronous mode

Table4.32 API Functions: [SCI/SCIF/RSCI in Asynchronous Mode]

API Function Name	Function
R_<Config_SCI0>_Create	Executes initialization processing that is required before controlling the SCI/SCIF asynchronous mode.
R_<Config_SCI0>_Start	Starts serial communications.
R_<Config_SCI0>_Stop	Stops serial communications.
R_<Config_SCI0>_Serial_Send	Starts transmission. (Asynchronous mode)
R_<Config_SCI0>_Serial_Receive	Starts reception. (Asynchronous mode)
R_<Config_SCI0>_Serial_Multiprocessor_Send	[SCI/RSCI] Starts transmission. (Multi-processor communications function)
R_<Config_SCI0>_Serial_Multiprocessor_Receive	[SCI/RSCI] Starts reception. (Multi-processor communications function)
R_<Config_SCI0>_Create_UserInit	Executes user-specific initialization processing for the SCI/SCIF asynchronous mode.
r_<Config_SCI0>_transmitend_interrupt	[SCI/RSCI] Executes processing in response to transmit end interrupts.
r_<Config_SCI0>_transmit_interrupt	[SCI/RSCI] Executes processing in response to transmit data empty interrupts.
r_<Config_SCI0>_receive_interrupt	[SCI/RSCI] Executes processing in response to receive data full interrupts.
r_<Config_SCI0>_receiveerror_interrupt	[SCI/RSCI] Executes processing in response to receive error interrupts.
r_<Config_SCI0>_teif_interrupt	[SCIF] Executes processing in response to transmit end interrupts.
r_<Config_SCI0>_txif_interrupt	[SCIF] Executes processing in response to transmit FIFO data empty interrupts.
r_<Config_SCI0>_rxif_interrupt	[SCIF] Executes processing in response to receive FIFO data full interrupts.
r_<Config_SCI0>_drif_interrupt	[SCIF] Executes processing in response to receive data ready interrupts.
r_<Config_SCI0>_erif_interrupt	[SCIF] Executes processing in response to receive error interrupts.
r_<Config_SCI0>_brif_interrupt	[SCIF] Executes processing in response to break interrupts.
r_<Config_SCI0>_callback_transmitend	[SCI/RSCI] Executes processing in response to transmit end interrupts or transmit data empty interrupts. [SCIF] Executes processing in response to transmit end interrupts or transmit FIFO data empty interrupts.
r_<Config_SCI0>_callback_receiveend	[SCI/RSCI] Executes processing in response to receive data full interrupts. [SCIF] Executes processing in response to receive FIFO data full interrupts.
r_<Config_SCI0>_callback_receiveerror	[SCI/RSCI] Executes processing in response to receive error interrupts.
r_<Config_SCI0>_callback_error	[SCIF] Executes processing in response to receive error interrupts or break interrupts.

R_<Config_SCI0>_Create

This API function executes initialization processing that is required before controlling the SCI/SCIF asynchronous mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_SCI0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Start

This API function starts serial communications.

[Syntax]

```
void        R_<Config_SCI0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Stop

This API function stops serial communications.

[Syntax]

```
void        R_<Config_SCI0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Serial_Send

This API function starts transmission. (Asynchronous mode)

Remark1. This API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. Calling R_<Config_SCI0>_Start is required before this API function is called to execute transmission.

[Syntax]

MD_STATUS	R_<Config_SCI0>_Serial_Send (uint8_t * const *tx_buf*, uint16_t *tx_num*);

[Argument(s)]

I/O	Argument	Description
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_SCI0>_Serial_Receive

This API function starts reception. (Asynchronous mode)

Remark1. This API function repeats the reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark2. Calling R_<Config_SCI0>_Start is required before this API function is called to execute reception.

[Syntax]

```
MD_STATUS     R_<Config_SCI0>_Serial_Receive ( uint8_t * const rx_buf, uint16_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
O	uint8_t * const *rx_buf*;	Pointer to the buffer where the received data are to be stored
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *rx_num* is invalid.

R_<Config_SCI0>_Serial_Multiprocessor_Send

[SCI/RSCI] This API function starts transmission. (Multi-processor communications function)

Remark1. In multi-processor communications, a transmission cycle consists of an ID transmission cycle to specify a receiving station, and a data transmission cycle to transmit data to the specified receiving station.

Remark2. In the ID transmission cycle, this API function repeats the transmission of single bytes of data from the buffer specified by the argument *id_buf* the number of times specified by the argument *id_num*.

Remark3. In the data transmission cycle, this API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark4. Calling R_<Config_SCI0>_Start is required before this API function is called to execute transmission.

[Syntax]

MD_STATUS R_<Config_SCI0>_Serial_Multiprocessor_Send (uint8_t * const *id_buf*, uint16_t *id_num*, uint8_t * const *tx_buf*, uint16_t *tx_num*);

[Argument(s)]

I/O	Argument	Description
I	uint8_t * const *id_buf*;	Pointer to the buffer where the IDs to be transmitted are stored
I	uint16_t *id_num*;	Number of IDs to be transmitted
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_SCI0>_Serial_Multiprocessor_Receive

[SCI/RSCI] This API function starts reception. (Multi-processor communications function)

Remark1. In multi-processor communications, a receiving station with an ID matching a received ID receives the data that follows the ID.

Remark2. This API function repeats the reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark3. Calling R_<Config_SCI0>_Start is required before this API function is called to execute reception.

[Syntax]

```
MD_STATUS R_<Config_SCI0>_Serial_Multiprocessor_Receive ( uint8_t * const rx_buf,
uint16_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
O	uint8_t * const *rx_buf*;	Pointer to the buffer where the received data are to be stored
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *rx_num* is invalid.

R_<Config_SCI0>_Create_UserInit

This API function executes user-specific initialization processing for the SCI/SCIF asynchronous mode.

Remark This API function is called from R_<Config_SCI0>_Start as a callback routine.

[Syntax]

```
void R_<Config_SCI0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_transmitend_interrupt

[SCI/RSCI] This API function executes processing in response to transmit end interrupts.

[Syntax]

```
void        r_<Config_SCI0>_transmitend_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_transmit_interrupt

[SCI/RSCI] This API function executes processing in response to transmit data empty interrupts.

[Syntax]

```
void        r_<Config_SCI0>_transmit_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_receive_interrupt

[SCI/RSCI] This API function executes processing in response to receive data full interrupts.

Receive interrupt request (SCI0.SCR.BIT.RIE=0 / RSCI10.SCR0.BIT.RIE=0) will be disabled inside this handler. If this caused any problem, please edit the source code at your discretion.

[Syntax]

```
void        r_<Config_SCI0>_receive_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_receiveerror_interrupt

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

[Syntax]

```
void        r_<Config_SCI0>_receiveerror_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_teif_interrupt

[SCIF] This API function executes processing in response to transmit end interrupts.

[Syntax]

```
void        r_<Config_SCI0>_teif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_txif_interrupt

[SCIF] This API function executes processing in response to transmit FIFO data empty interrupts.

[Syntax]

```
void        r_<Config_SCI0>_txif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_rxif_interrupt

[SCIF] This API function executes processing in response to receive FIFO data full interrupts.

[Syntax]

```
void        r_<Config_SCI0>_rxif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_drif_interrupt

[SCIF] This API function executes processing in response to receive data ready interrupts.

[Syntax]

```
void        r_<Config_SCI0>_drif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_erif_interrupt

[SCIF] This API function executes processing in response to receive error interrupts.

Remark This API function is called as the interrupt handler for interrupts due to framing errors or parity errors.

[Syntax]

```
void r_<Config_SCI0>_erif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_brif_interrupt

[SCIF] This API function executes processing in response to break interrupts.

Remark This API function is called as the interrupt handler for interrupts due to break signal detection or overrun errors.

[Syntax]

```
void r_<Config_SCI0>_brif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_transmitend

[SCI/RSCI] This API function executes processing in response to transmit end interrupts or transmit data empty interrupts.

Remark1. This API function is called from r_<Config_SCI0>_transmitend_interrupt or r_<Config_SCI0>_transmit_interrupt as a callback routine.

[SCIF] This API function executes processing in response to transmit end interrupts or transmit FIFO data empty interrupts.

Remark2. This API function is called from r_<Config_SCI0>_teif_interrupt or r_<Config_SCI0>_txif_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_SCI0>_callback_transmitend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_receiveend

[SCI/RSCI] This API function executes processing in response to receive data full interrupts.
Remark1. This API function is called from r_<Config_SCI0>_receive_interrupt as a callback routine.

[SCIF] This API function executes processing in response to receive FIFO data full interrupts.
Remark2. This API function is called from r_<Config_SCI0>_rxif_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_SCI0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_receiveerror

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

Remark This API function is called from r_<Config_SCI0>_receiveerror_interrupt as a callback routine.

[Syntax]

```
void r_<Config_SCI0>_callback_receiveerror ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_error

[SCIF] This API function executes processing in response to receive error interrupts or break interrupts.

Remark This API function is called from r_<Config_SCI0>_erif_interrupt or r_<Config_SCI0>_brif_interrupt as a callback routine.

[Syntax]

```
void r_<Config_SCI0>_callback_error ( scif_error_type_t error_type );
```

[Argument(s)]

I/O	Argument	Description
I	scif_error_type_t *error_type*;	Interrupt sources RECEIVE_ERROR: Framing error or parity error OVERRUN_ERROR: Overrun error BREAK_DETECT: Break signal detection

[Return value]

None.

Usage example

Transmitting the upper-case character "A":

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t g_sci0_tx_buf;
void main(void)
{
    /* Start the SCI0 channel */
    R_Config_SCI0_Start();

    /* Transmit SCI0 data */
    R_Config_SCI0_Serial_Send((uint8_t *)&g_sci0_tx_buf, 1U);

    while (1U)
    {
        nop();
    }
}
```

Config_SCI0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_sci0_tx_buf;
/* End user code. Do not edit comment generated here */

void R_Config_SCI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_sci0_tx_buf = 'A';
    /* End user code. Do not edit comment generated here */
}
static void r_Config_SCI0_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI0_callback_transmitend. Do not edit comment generated
here */
    /* Stop the SCI0 channel */
    R_Config_SCI0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.32 SCI/SCIF Clock Synchronous Mode

The Code Generator outputs the following API functions for the SCI/SCIF clock synchronous mode.

Table4.33 API Functions: [SCI/SCIF/RSCI Clock Synchronous Mode]

API Function Name	Function
R_<Config_SCI0>_Create	Executes initialization processing that is required before controlling the SCI/SCIF clock synchronous mode.
R_<Config_SCI0>_Start	Starts serial communications.
R_<Config_SCI0>_Stop	Stops serial communications.
R_<Config_SCI0>_Serial_Send	Starts transmission. (Clock synchronous mode)
R_<Config_SCI0>_Serial_Receive	Starts reception. (Clock synchronous mode)
R_<Config_SCI0>_Serial_Send_Receive	Starts transmission and reception. (Clock synchronous mode)
R_<Config_SCI0>_Create_UserInit	Executes user-specific initialization processing for the SCI/SCIF clock synchronous mode.
r_<Config_SCI0>_transmitend_interrupt	[SCI/RSCI] Executes processing in response to transmit end interrupts.
r_<Config_SCI0>_transmit_interrupt	[SCI/RSCI] Executes processing in response to transmit data empty interrupts.
r_<Config_SCI0>_receive_interrupt	[SCI/RSCI] Executes processing in response to receive data full interrupts.
r_<Config_SCI0>_receiveerror_interrupt	[SCI/RSCI] Executes processing in response to receive error interrupts.
r_<Config_SCI0>_teif_interrupt	[SCIF] Executes processing in response to transmit end interrupts.
r_<Config_SCI0>_txif_interrupt	[SCIF] Executes processing in response to transmit FIFO data empty interrupts.
r_<Config_SCI0>_rxif_interrupt	[SCIF] Executes processing in response to receive FIFO data full interrupts.
r_<Config_SCI0>_erif_interrupt	[SCIF] Executes processing in response to receive error interrupts.
r_<Config_SCI0>_brif_interrupt	[SCIF] Executes processing in response to break interrupts.
r_<Config_SCI0>_callback_transmitend	[SCI/RSCI] Executes processing in response to transmit end interrupts or transmit data empty interrupts. [SCIF] Executes processing in response to transmit end interrupts or transmit FIFO data empty interrupts.
r_<Config_SCI0>_callback_receiveend	[SCI/RSCI] Executes processing in response to receive data full interrupts. [SCIF] Executes processing in response to receive FIFO data full interrupts.
r_<Config_SCI0>_callback_receiveerror	[SCI/RSCI] Executes processing in response to receive error interrupts.
r_<Config_SCI0>_callback_error	[SCIF] Executes processing in response to receive error interrupts or break interrupts.

R_<Config_SCI0>_Create

This API function executes initialization processing that is required before controlling the SCI/SCIF clock synchronous mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_<Config_SCI0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Start

This API function starts serial communications.

[Syntax]

```
void        R_<Config_SCI0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Stop

This API function stops serial communications.

[Syntax]

```
void        R_<Config_SCI0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Serial_Send

This API function starts transmission. (Clock synchronous mode)

Remark1. This API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. Calling R_<Config_SCI0>_Start is required before this API function is called to execute transmission.

[Syntax]

MD_STATUS	R_<Config_SCI0>_Serial_Send (uint8_t * const *tx_buf*, uint16_t *tx_num*);

[Argument(s)]

I/O	Argument	Description
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_SCI0>_Serial_Receive

This API function starts reception. (Clock synchronous mode)

Remark1. This API function repeats the reception of single bytes of data the number of times specified by the artgument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark2. Calling R_<Config_SCI0>_Start is required before this API function is called to execute reception.

[Syntax]

```
MD_STATUS      R_<Config_SCI0>_Serial_Receive ( uint8_t * const rx_buf, uint16_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
O	uint8_t * const *rx_buf*;	Pointer to the buffer where the received data are to be stored
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *rx_num* is invalid.

R_<Config_SCI0>_Serial_Send_Receive

This API function starts transmission and reception. (Clock synchronous mode)

Remark1. To transmit data, this API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. To receive data, this API function repeats the reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark3. Calling R_<Config_SCI0>_Start is required before this API function is called to execute reception.

[Syntax]

MD_STATUS R_<Config_SCI0>_Serial_Send_Receive (uint8_t * const *tx_buf*, uint16_t *tx_num*, uint8_t * const *rx_buf*, uint16_t *rx_num*);

[Argument(s)]

I/O	Argument	Description
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted
O	uint8_t * const *rx_buf*;	Pointer to the buffer where the received data are to be stored
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_SCI0>_Create_UserInit

This API function executes user-specific initialization processing for the SCI/SCIF clock synchronous mode.

Remark This API function is called from R_<Config_SCI0>_Create as a callback routine.

[Syntax]

```
void    R_<Config_SCI0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_transmitend_interrupt

[SCI/RSCI] This API function executes processing in response to transmit end interrupts.

[Syntax]

```
void        r_<Config_SCI0>_transmitend_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_transmit_interrupt

[SCI/RSCI] This API function executes processing in response to transmit data empty interrupts.

[Syntax]

```
void        r_<Config_SCI0>_transmit_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_receive_interrupt

[SCI/RSCI] This API function executes processing in response to receive data full interrupts.

Receive interrupt request (SCI0.SCR.BIT.RIE=0 / RSCI10.SCR0.BIT.RIE=0) will be disabled inside this handler. If this caused any problem, please edit the source code at your discretion.

[Syntax]

```
void        r_<Config_SCI0>_receive_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_receiveerror_interrupt

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

[Syntax]

```
void        r_<Config_SCI0>_receiveerror_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_teif_interrupt

[SCIF] This API function executes processing in response to transmit end interrupts.

[Syntax]

```
void        r_<Config_SCI0>_teif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_txif_interrupt

[SCIF] This API function executes processing in response to transmit FIFO data empty interrupts.

[Syntax]

```
void        r_<Config_SCI0>_txif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_rxif_interrupt

[SCIF] This API function executes processing in response to receive FIFO data full interrupts.

[Syntax]

```
void        r_<Config_SCI0>_rxif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_erif_interrupt

[SCIF] This API function executes processing in response to receive error interrupts.

Remark This API function is called as the interrupt handler for interrupts due to framing errors or parity errors.

[Syntax]

```
void r_<Config_SCI0>_erif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_brif_interrupt

[SCIF] This API function executes processing in response to break interrupts.

Remark This API function is called as the interrupt handler for interrupts due to break signal detection or overrun errors.

[Syntax]

```
void r_<Config_SCI0>_brif_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_transmitend

[SCI/RSCI] This API function executes processing in response to transmit end interrupts or transmit data empty interrupts.

Remark1. This API function is called from r_<Config_SCI0>_transmitend_interrupt or r_<Config_SCI0>_transmit_interrupt as a callback routine.

[SCIF] This API function executes processing in response to transmit end interrupts or transmit FIFO data empty interrupts.

Remark2. This API function is called from r_<Config_SCI0>_teif_interrupt or r_<Config_SCI0>_txif_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_SCI0>_callback_transmitend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_receiveend

[SCI/RSCI] This API function executes processing in response to receive data full interrupts.
Remark1. This API function is called from r_<Config_SCI0>_receive_interrupt as a callback routine.

[SCIF] This API function executes processing in response to receive FIFO data full interrupts.
Remark2. This API function is called from r_<Config_SCI0>_rxif_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_SCI0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_receiveerror

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

Remark This API function is called from r_<Config_SCI0>_receiveerror_interrupt as a callback routine.

[Syntax]

```
void r_<Config_SCI0>_callback_receiveerror ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_error

[SCIF] This API function executes processing in response to receive error interrupts or break interrupts.

Remark This API function is called from r_<Config_SCI0>_erif_interrupt or r_<Config_SCI0>_brif_interrupt as a callback routine.

[Syntax]

```
void r_<Config_SCI0>_callback_error ( scif_error_type_t error_type );
```

[Argument(s)]

I/O	Argument	Description
I	scif_error_type_t *error_type*;	Interrupt sources RECEIVE_ERROR: Framing error or parity error OVERRUN_ERROR: Overrun error BREAK_DETECT: Break signal detection

[Return value]

None.

Usage example

Transmitting received data:

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t g_sci0_tx_buf;
extern volatile uint8_t g_sci0_rx_buf;
void main(void)
{
    /* Start the SCI0 channel */
    R_Config_SCI0_Start();

    /* Transmit and receive SCI0 data */
    R_Config_SCI0_Serial_Send_Receive((uint8_t *)&g_sci0_tx_buf, 1U, (uint8_t *)&g_sci0_rx_buf, 1U);

    while (1U)
    {
        nop();
    }
}
```

Config_SCI0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_sci0_tx_buf;
volatile uint8_t g_sci0_rx_buf;
/* End user code. Do not edit comment generated here */

void R_Config_SCI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_sci0_tx_buf = 0U;
    g_sci0_rx_buf = 0U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_transmitend(void)
{
    /* Start user code for r_Config_SCI0_callback_transmitend. Do not edit comment generated here */
    /* Transmit and receive SCI0 data */
    R_Config_SCI0_Serial_Send_Receive((uint8_t *)&g_sci0_tx_buf, 1U, (uint8_t *)&g_sci0_rx_buf, 1U);
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_receiveend(void)
{
    /* Start user code for r_Config_SCI0_callback_receiveend. Do not edit comment generated here */
    g_sci0_tx_buf = g_sci0_rx_buf;
    g_sci0_rx_buf = 0U;
    /* End user code. Do not edit comment generated here */
}
```

4.2.33 Single Scan Mode S12AD

The Code Generator outputs the following API functions for the single scan mode S12AD.

Table4.34 API Functions: [Single Scan Mode S12AD]

API Function Name	Function
R_<Config_S12AD0>_Create	Executes initialization processing that is required before controlling the single scan mode S12AD.
R_<Config_S12AD0>_Start	Starts A/D conversion.
R_<Config_S12AD0>_Stop	Stops A/D conversion.
R_<Config_S12AD0>_Get_ValueResult	Gets the result of conversion.
R_<Config_S12AD0>_Set_CompareValue	Sets the compare level.
R_<Config_S12AD0>_Set_CompareAValue	Sets the compare level for window A.
R_<Config_S12AD0>_Set_CompareBValue	Sets the compare level for window B.
R_<Config_S12AD0>_Create_UserInit	Executes user-specific initialization processing for the single scan mode S12AD.
r_<Config_S12AD0>_interrupt	Executes processing in response to scan end interrupts.
r_<Config_S12AD0>_compare_interrupt	Executes processing in response to compare interrupts.
r_<Config_S12AD0>_compare_interruptA	Executes processing in response to compare interrupts for window A.
r_<Config_S12AD0>_compare_interruptB	Executes processing in response to compare interrupts for window B.

R_<Config_S12AD0>_Create

This API function executes initialization processing that is required before controlling the single scan mode S12AD.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_S12AD0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Start

This API function starts A/D conversion.

[Syntax]

```
void        R_<Config_S12AD0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Stop

This API function stops A/D conversion.

[Syntax]

```
void        R_<Config_S12AD0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_S12AD0>_Get_ValueResult

This API function gets the result of conversion.

[Syntax]

```
void    R_<Config_S12AD0>_Get_ValueResult ( ad_channel_t channel, uint16_t * const buffer );
```

[Argument(s)]

For RX130 or RX230/RX231

I/O	Argument	Description
I	ad_channel_t *channel*;	Channel number ADCHANNEL0: Input channel AN000 ADCHANNEL1: Input channel AN001 ADCHANNEL2: Input channel AN002 ADCHANNEL3: Input channel AN003 ADCHANNEL4: Input channel AN004 ADCHANNEL5: Input channel AN005 ADCHANNEL6: Input channel AN006 ADCHANNEL7: Input channel AN007 ADCHANNEL16: Input channel AN016 ADCHANNEL17: Input channel AN017 ADCHANNEL18: Input channel AN018 ADCHANNEL19: Input channel AN019 ADCHANNEL20: Input channel AN020 ADCHANNEL21: Input channel AN021 ADCHANNEL22: Input channel AN022 ADCHANNEL23: Input channel AN023 ADCHANNEL24: Input channel AN024 ADCHANNEL25: Input channel AN025 ADCHANNEL26: Input channel AN026 ADCHANNEL27: Input channel AN027 ADCHANNEL28: Input channel AN028 ADCHANNEL29: Input channel AN029 ADCHANNEL30: Input channel AN030 ADCHANNEL31: Input channel AN031 ADTEMPSENSOR: Extended analog input (temperature sensor output) ADINTERREFVOLT: Extended analog input (internal reference voltage) ADSELFDIAGNOSIS: Result of self-diagnosis ADDATADUPLICATION: Double-trigger mode result
O	uint16_t * const *buffer*;	Pointer to the location where the acquired results of conversion are to be stored

For other devices

I/O	Argument	Description
I	ad_channel_t *channel*;	Channel number ADCHANNEL0: Input channel AN000 ADCHANNEL1: Input channel AN001 ADCHANNEL2: Input channel AN002 ADCHANNEL3: Input channel AN003 ADCHANNEL4: Input channel AN004 ADCHANNEL5: Input channel AN005 ADCHANNEL6: Input channel AN006 ADCHANNEL7: Input channel AN007 ADCHANNEL8: Input channel AN008 ADCHANNEL9: Input channel AN009 ADCHANNEL10: Input channel AN010 ADCHANNEL11: Input channel AN011 ADCHANNEL12: Input channel AN012 ADCHANNEL13: Input channel AN013 ADCHANNEL14: Input channel AN014 ADCHANNEL15: Input channel AN015 ADCHANNEL16: Input channel AN016 ADCHANNEL17: Input channel AN017 ADCHANNEL18: Input channel AN018 ADCHANNEL19: Input channel AN019 ADCHANNEL20: Input channel AN020 ADTEMPSENSOR: Extended analog input (temperature sensor output) ADINTERREFVOLT: Extended analog input (internal reference voltage) ADSELFDIAGNOSIS: Result of self-diagnosis ADDATADUPLICATION: Double-trigger mode result ADDATADUPLICATIONA: Double-trigger mode A result ADDATADUPLICATIONB: Double-trigger mode B result
O	uint16_t * const *buffer*;	Pointer to the location where the acquired results of conversion are to be stored

[Return value]

None.

R_<Config_S12AD0>_Set_CompareValue

This API function sets the compare level.

[Syntax]

```
void    R_<Config_S12AD0>_Set_CompareValue ( uint16_t reg_value0, uint16_t reg_value1);
```

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Set_CompareAValue

This API function sets the compare level for window A.

[Syntax]

```
void    R_<Config_S12AD0>_Set_CompareAValue ( uint16_t reg_value0, uint16_t reg_value1);
```

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Set_CompareBValue

This API function sets the compare level for window B.

[Syntax]

void R_<Config_S12AD0>_Set_CompareBValue (uint16_t *reg_value0*, uint16_t *reg_value1*);

[Argument(s)]

I/O	Argument	Description
I	uint16_t *reg_value0*;	Value to be set in compare register 0
I	uint16_t *reg_value1*;	Value to be set in compare register 1

[Return value]

None.

R_<Config_S12AD0>_Create_UserInit

This API function executes user-specific initialization processing for the S12AD in single scan mode.

Remark This API function is called from R_<Config_S12AD0>_Create as a callback routine.

[Syntax]

```
void R_<Config_S12AD0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_interrupt

This API function executes processing in response to scan end interrupts.

[Syntax]

```
void        r_<Config_S12AD0>_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interrupt

This API function executes processing in response to compare interrupts.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interruptA

This API function executes processing in response to compare interrupts for window A.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interruptA ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_S12AD0>_compare_interruptB

This API function executes processing in response to compare interrupts for window B.

[Syntax]

```
void        r_<Config_S12AD0>_compare_interruptB ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Getting the result of A/D conversion:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the AD0 converter */
    R_Config_S12AD0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_S12AD0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t g_s12ad0_ch000_value;
/* End user code. Do not edit comment generated here */

static void r_Config_S12AD0_interrupt(void)
{
    /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, (uint16_t *)&g_s12ad0_ch000_value);
    /* End user code. Do not edit comment generated here */
}
```

4.2.34 Smart Card Interface Mode

The Code Generator outputs the following API functions for the smart card interface mode.

Table4.35 API Functions: [SCI/RSCI in Smart Card Interface Mode]

API Function Name	Function
R_<Config_SCI0>_Create	Executes initialization processing that is required before controlling the smart card interface mode.
R_<Config_SCI0>_Start	Starts serial communications.
R_<Config_SCI0>_Stop	Stops serial communications.
R_<Config_SCI0>_SmartCard_Send	Starts transmission. (Smart card interface mode)
R_<Config_SCI0>_SmartCard_Receive	Starts reception. (Smart card interface mode)
R_<Config_SCI0>_Create_UserInit	Executes user-specific initialization processing for the smart card interface mode.
r_<Config_SCI0>_transmit_interrupt	Executes processing in response to transmit data empty interrupts.
r_<Config_SCI0>_receive_interrupt	Executes processing in response to receive data full interrupts.
r_<Config_SCI0>_receiveerror_interrupt	Executes processing in response to receive error interrupts.
r_<Config_SCI0>_callback_transmitend	Executes processing in response to transmit data empty interrupts.
r_<Config_SCI0>_callback_receiveend	Executes processing in response to receive data full interrupts.
r_<Config_SCI0>_callback_receiveerror	Executes processing in response to receive error interrupts.

R_<Config_SCI0>_Create

This API function executes initialization processing that is required before controlling the smart card interface.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_SCI0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Start

This API function starts serial communications.

[Syntax]

```
void        R_<Config_SCI0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_Stop

This API function stops serial communications.

[Syntax]

```
void        R_<Config_SCI0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_SCI0>_SmartCard_Send

This API function starts transmission. (Smart card interface mode)

Remark1. This API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. Calling R_<Config_SCI0>_Start is required before this API function is called to execute transmission.

[Syntax]

```
MD_STATUS      R_<Config_SCI0>_SmartCard_Send ( uint8_t * const tx_buf, uint16_t tx_num );
```

[Argument(s)]

I/O	Argument	Description
I	uint8_t * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_SCI0>_SmartCard_Receive

This API function starts reception. (Smart card interface mode)

Remark1. This API function repeats the reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark2. Calling R_<Config_SCI0>_Start is required before this API function is called to execute reception.

[Syntax]

```
MD_STATUS    R_<Config_SCI0>_SmartCard_Receive ( uint8_t * const rx_buf, uint16_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
O	uint8_t * const *rx_buf*;	Pointer to the buffer where the received data are to be stored
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *rx_num* is invalid.

R_<Config_SCI0>_Create_UserInit

This API function executes user-specific initialization processing for the smart card interface.

Remark This API function is called from R_<Config_SCI0>_Create as a callback routine.

[Syntax]

```
void R_<Config_SCI0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_transmit_interrupt

This API function executes processing in response to transmit data empty interrupts.

[Syntax]

```
void        r_<Config_SCI0>_transmit_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_receive_interrupt

This API function executes processing in response to receive data full interrupts.

[Syntax]

```
void        r_<Config_SCI0>_receive_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_receiveerror_interrupt

This API function executes processing in response to receive error interrupts.

[Syntax]

```
void        r_<Config_SCI0>_receiveerror_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_transmitend

This API function executes processing in response to transmit data empty interrupts.

Remark his API function is called from r_<Config_SCI0>_transmit_interrupt as a callback routine.

[Syntax]

```
void r_<Config_SCI0>_callback_transmitend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_receiveend

This API function executes processing in response to receive data full interrupts.

Remark This API function is called from r_<Config_SCI0>_receive_interrupt as a callback routine.

[Syntax]

```
void r_<Config_SCI0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_SCI0>_callback_receiveerror

This API function executes processing in response to receive error interrupts.

Remark this API function is called from r_<Config_SCI0>_receiveerror_interrupt as a callback routine.

[Syntax]

```
void r_<Config_SCI0>_callback_receiveerror ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Transmitting one byte, and then switching to the reception of one byte:

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t g_sci0_tx_buf;
void main(void)
{
    /* Start the SCI0 channel */
    R_Config_SCI0_Start();

    /* Transmit SCI0 data */
    R_Config_SCI0_SmartCard_Send((uint8_t *)&g_sci0_tx_buf, 1U);

    while (1U)
    {
        nop();
    }
}
```

Config_SCI0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t g_sci0_tx_buf;
volatile uint8_t g_sci0_rx_buf;
/* End user code. Do not edit comment generated here */

void R_Config_SCI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_sci0_tx_buf = 'A';
    /* End user code. Do not edit comment generated here */
}

static void r_Config_SCI0_callback_transmitend(void)
{
    /* Start user code for r_<onfig_SCI0_callback_transmitend. Do not edit comment generated
here */
    /* Stop the SCI0 channel */
    R_Config_SCI0_Stop();

    /* Initializes SCI0 */
    R_Config_SCI0_Create();

    /* Start the SCI0 channel */
    R_Config_SCI0_Start();

    /* Receive SCI0 data */
    R_Config_SCI0_SmartCard_Receive((uint8_t *)&g_sci0_rx_buf, 1U);
    /* End user code. Do not edit comment generated here */
}
static void r_Config_SCI0_callback_receiveend(void)
{
    /* Start user code for r_Config_SCI0_callback_receiveend. Do not edit comment generated here
*/
    /* Stop the SCI0 channel */
    R_Config_SCI0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.35 SPI Clock Synchronous Mode

The Code Generator outputs the following API functions for the SPI clock synchronous mode (RSPI/SCI/RSCI).

Table4.36 API Functions: [RSPI/SCI/RSCI in SPI Clock Synchronous Mode]

API Function Name	Function
R_<Config_RSPI0>_Create	Executes initialization processing that is required before controlling the SPI clock synchronous mode (RSPI/SCI/RSCI).
R_<Config_RSPI0>_Start	Starts serial communications.
R_<Config_RSPI0>_Stop	Stops serial communications.
R_<Config_RSPI0>_Send	[RSPI] Starts transmission.
R_<Config_RSPI0>_Send_Receive	[RSPI] Starts transmission and reception.
R_<Config_RSPI0>_SPI_Master_Send	[SCI/RSCI] Starts master transmission. (Simple SPI mode)
R_<Config_RSPI0>_SPI_Master_Send_Receive	[SCI/RSCI] Starts master transmission and reception. (Simple SPI mode)
R_<Config_RSPI0>_SPI_Slave_Send	[SCI/RSCI] Starts slave transmission. (Simple SPI mode)
R_<Config_RSPI0>_SPI_Slave_Send_Receive	[SCI/RSCI] Starts slave transmission and reception. (Simple SPI mode)
R_<Config_RSPI0>_Create_UserInit	Executes user-specific initialization processing for the SPI clock synchronous mode (RSPI/SCI/RSCI).
r_<Config_RSPI0>_receive_interrupt	Executes processing in response to receive buffer full interrupts.
r_<Config_RSPI0>_transmit_interrupt	Executes processing in response to transmit buffer empty interrupts.
r_<Config_RSPI0>_error_interrupt	[RSPI] Executes processing in response to error interrupts.
r_<Config_RSPI0>_idle_interrupt	[RSPI] Executes processing in response to RSPI idle interrupts.
r_<Config_RSPI0>_transmitend_interrupt	[SCI/RSCI] Executes processing in response to transmit end interrupts.
r_<Config_RSPI0>_receiveerror_interrupt	[SCI/RSCI] Executes processing in response to receive error interrupts.
r_<Config_RSPI0>_callback_receiveend	Executes processing in response to receive buffer full interrupts.
r_<Config_RSPI0>_callback_transmitend	Executes processing in response to transmit buffer empty interrupts. [RSPI] Executes processing in response to RSPI idle interrupts. [SCI/RSCI] Executes processing in response to transmit end interrupts.
r_<Config_RSPI0>_callback_error	[RSPI] Executes processing in response to error interrupts.
r_<Config_RSPI0>_callback_receiveerror	[SCI/RSCI] Executes processing in response to receive error interrupts.

R_<Config_RSPI0>_Create

This API function executes initialization processing that is required before controlling the clock synchronous SPI mode (RSPI/SCI/RSCI).

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void    R_<Config_RSPI0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RSPI0>_Start

This API function starts serial communications.

[Syntax]

```
void        R_<Config_RSPI0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RSPI0>_Stop

This API function stops serial communications.

[Syntax]

```
void        R_<Config_RSPI0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RSPI0>_Send

[RSPI] This API function starts transmission.

Remark1. This API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute transmission.

[Syntax]

MD_STATUS R_<Config_RSPI0>_Send (type * const *tx_buf*, uint16_t *tx_num*);

[Argument(s)]

I/O	Argument	Description
I	type * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_RSPI0>_Send_Receive

[RSPI] This API function starts transmission and reception.

Remark1. To transmit data, this API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. To receive data, this API function repeats the reception of single bytes of data the number of times specified by the argument *tx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark3. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute transmission and reception.

[Syntax]

```
MD_STATUS      R_<Config_RSPI0>_Send_Receive ( type * const tx_buf, uint16_t tx_num,
 type * const rx_buf );
```

[Argument(s)]

I/O	Argument	Description
I	type * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *tx_num*;	Number of bytes to be transmitted and received
O	type * const *rx_buf*;	Pointer to the buffer where the received data are to be stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_RSPI0>_SPI_Master_Send

[SCI/RSCI] This API function starts master transmission. (Simple SPI mode)

Remark1. his API function repeats the master transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute master transmission.

[Syntax]

```
MD_STATUS      R_<Config_RSPI0>_SPI_Master_Send (type * const tx_buf, uint16_t tx_num );
```

[Argument(s)]

I/O	Argument	Description
I	type * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_RSPI0>_SPI_Master_Send_Receive

[SCI/RSCI] This API function starts master transmission and reception. (Simple SPI mode)

Remark1. To transmit data as a master, this API function repeats the master transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. To receive data as a master, this API function repeats the master reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark3. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute transmission and reception.

[Syntax]

MD_STATUS R_<Config_RSPI0>_SPI_Master_Send_Receive (type * const *tx_buf*, uint16_t *tx_num*, type * const *rx_buf*, uint16_t *rx_num*);

[Argument(s)]

I/O	Argument	Description
I	type * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *tx_num*;	Number of bytes to be transmitted
O	type * const *rx_buf*;	Pointer to the buffer where the received data are to be stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_RSPI0>_SPI_Slave_Send

[SCI/RSCI] This API function starts slave transmission. (Simple SPI mode)

Remark1. This API function repeats the slave transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute slave transmission.

[Syntax]

```
MD_STATUS      R_<Config_RSPI0>_SPI_Slave_Send (type * const tx_buf, uint16_t tx_num );
```

[Argument(s)]

I/O	Argument	Description
I	type * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_RSPI0>_SPI_Slave_Send_Receive

[SCI/RSCI] This API function starts slave transmission and reception. (Simple SPI mode)

Remark1. To transmit data as a slave, this API function repeats the slave transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. To receive data as a slave, this API function repeats the slave reception of single bytes of data the number of times specified by the argument *rx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark3. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute transmission and reception.

[Syntax]

```
MD_STATUS      R_<Config_RSPI0>_SPI_Slave_Send_Receive ( type * const tx_buf,
 uint16_t tx_num, type * const rx_buf, uint16_t rx_num );
```

[Argument(s)]

I/O	Argument	Description
I	type * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *tx_num*;	Number of bytes to be transmitted
O	type * const *rx_buf*;	Pointer to the buffer where the received data are to be stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *rx_num*;	Number of bytes to be received

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_RSPI0>_Create_UserInit

This API function executes user-specific initialization processing for the clock synchronous SPI mode (RSPI/SCI/RSCI).

Remark This API function is called from R_<Config_RSPI0>_Create as a callback routine.

[Syntax]

```
void R_<Config_RSPI0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_receive_interrupt

This API function executes processing in response to receive buffer full interrupts.

Receive interrupt request (SCI0.SCR.BIT.RIE=0 / RSCI10.SCR0.BIT.RIE=0) or Receive Buffer full interrupt request (RSPI0.SPCR.BIT.SPRIE = 0) will be disabled inside this handler. If this caused any problem, please edit the source code at your disretion.

[Syntax]

```
void        r_<Config_RSPI0>_receive_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_transmit_interrupt

This API function executes processing in response to transmit buffer empty interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_transmit_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_error_interrupt

[RSPI] This API function executes processing in response to error interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_error_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_idle_interrupt

[RSPI] This API function executes processing in response to RSPI idle interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_idle_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_transmitend_interrupt

[SCI/RSCI] This API function executes processing in response to transmit end interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_transmitend_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_receiveerror_interrupt

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_receiveerror_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_callback_receiveend

This API function executes processing in response to receive buffer full interrupts.

Remark This API function is called from r_<Config_RSPI0>_receive_interrupt as a callback routine.

[Syntax]

```
void r_<Config_RSPI0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_callback_transmitend

This API function executes processing in response to transmit buffer empty interrupts.

Remark1. This API function is called from r_<Config_RSPI0>_transmit_interrupt as a callback routine.

[RSPI] This API function executes processing in response to RSPI idle interrupts.

Remark2. This API function is called from r_<Config_RSPI0>_idle_interrupt as a callback routine.

[SCI/RSCI] This API function executes processing in response to transmit end interrupts.

Remark3. his API function is called from r_<Config_RSPI0>_transmitend_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_RSPI0>_callback_transmitend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_callback_error

[RSPI] This API function executes processing in response to error interrupts.

Remark1. This API function is called from r_<Config_RSPI0>_error_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_RSPI0>_callback_error ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_callback_receiveerror

[SCI/RSCI] This API function executes processing in response to receive error interrupts.

Remark1. This API function is called from r_<Config_RSPI0>_receiveerror_interrupt as a callback routine.

[Syntax]

```
void        r_<Config_RSPI0>_callback_receiveerror ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Stopping communications after completing transmission and reception of the same number of bytes:

main.c

```
#include "r_smc_entry.h"
extern volatile uint32_t g_rspi0_tx_buf[4];
extern volatile uint32_t g_rspi0_rx_buf[4];
void main(void)
{
    /* Start the RSPI0 module operation */
    R_Config_RSPI0_Start();

    /* Send and receive RSPI0 data */
    R_Config_RSPI0_Send_Receive((uint32_t *)g_rspi0_tx_buf, 4U, (uint32_t *)g_rspi0_rx_buf);

    while (1U)
    {
        nop();
    }
}
```

Config_RSPI0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint32_t g_rspi0_tx_buf[4];
volatile uint32_t g_rspi0_rx_buf[4];
/* End user code. Do not edit comment generated here */

void R_Config_RSPI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_rspi0_tx_buf[0] = 0x000000FF;
    g_rspi0_tx_buf[1] = 0x0000FF00;
    g_rspi0_tx_buf[2] = 0x00FF0000;
    g_rspi0_tx_buf[3] = 0xFF000000;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_RSPI0_callback_receiveend(void)
{
    /* Start user code for r_Config_RSPI0_callback_receiveend. Do not edit comment generated
here */
    /* Stop the RSPI0 module operation */
    R_Config_RSPI0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.36 SPI Operation Mode

The Code Generator outputs the following API functions for the SPI operation mode.

Table4.37 API Functions: [SPI Operation Mode]

API Function Name	Function
R_<Config_RSPI0>_Create	Executes initialization processing that is required before controlling the SPI operation mode.
R_<Config_RSPI0>_Start	Starts serial communications.
R_<Config_RSPI0>_Stop	Stops serial communications.
R_<Config_RSPI0>_Send	Starts transmission.
R_<Config_RSPI0>_Send_Receive	Starts transmission and reception.
R_<Config_RSPI0>_Create_UserInit	Executes user-specific initialization processing for the SPI operation mode.
r_<Config_RSPI0>_receive_interrupt	Executes processing in response to receive buffer full interrupts.
r_<Config_RSPI0>_transmit_interrupt	Executes processing in response to transmit buffer empty interrupts.
r_<Config_RSPI0>_error_interrupt	Executes processing in response to error interrupts.
r_<Config_RSPI0>_idle_interrupt	Executes processing in response to RSPI idle interrupts.
r_<Config_RSPI0>_callback_receiveend	Executes processing in response to receive buffer full interrupts.
r_<Config_RSPI0>_callback_transmitend	Executes processing in response to transmit buffer empty interrupts or RSPI idle interrupts.
r_<Config_RSPI0>_callback_error	Executes processing in response to error interrupts.

R_<Config_RSPI0>_Create

This API function executes initialization processing that is required before controlling the SPI operation mode.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_RSPI0>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RSPI0>_Start

This API function starts serial communications.

[Syntax]

```
void        R_<Config_RSPI0>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RSPI0>_Stop

This API function stops serial communications.

[Syntax]

```
void        R_<Config_RSPI0>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_RSPI0>_Send

This API function starts transmission.

Remark1. This API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute transmission.

[Syntax]

MD_STATUS R_<Config_RSPI0>_Send (type * const *tx_buf*, uint16_t *tx_num*);

[Argument(s)]

I/O	Argument	Description
I	type * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *tx_num*;	Number of bytes to be transmitted

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_RSPI0>_Send_Receive

This API function starts transmission and reception.

Remark1. To transmit data, this API function repeats the transmission of single bytes of data from the buffer specified by the argument *tx_buf* the number of times specified by the argument *tx_num*.

Remark2. To receive data, this API function repeats the reception of single bytes of data the number of times specified by the argument *tx_num*, storing the data in the buffer specified by the argument *rx_buf*.

Remark3. Calling R_<Config_RSPI0>_Start is required before this API function is called to execute transmission and reception.

[Syntax]

```
MD_STATUS      R_<Config_RSPI0>_Send_Receive ( type * const tx_buf, uint16_t tx_num,
 type * const rx_buf );
```

[Argument(s)]

I/O	Argument	Description
I	type * const *tx_buf*;	Pointer to the buffer where the data to be transmitted are stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI
I	uint16_t *tx_num*;	Number of bytes to be transmitted and received
O	type * const *rx_buf*;	Pointer to the buffer where the received data are to be stored, the “type” can be “uint32_t”, “uint16_t” and “uint8_t” based on the buffer access width set on the GUI

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	The specification of argument *tx_num* is invalid.

R_<Config_RSPI0>_Create_UserInit

This API function executes user-specific initialization processing for the SPI operation mode.

Remark This API function is called from R_<Config_RSPI0>_Create as a callback routine.

[Syntax]

```
void R_<Config_RSPI0>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_receive_interrupt

This API function executes processing in response to receive buffer full interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_receive_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_transmit_interrupt

This API function executes processing in response to transmit buffer empty interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_transmit_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_error_interrupt

This API function executes processing in response to error interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_error_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_idle_interrupt

This API function executes processing in response to RSPI idle interrupts.

[Syntax]

```
void        r_<Config_RSPI0>_idle_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_callback_receiveend

This API function executes processing in response to receive buffer full interrupts.

Remark This API function is called from r_<Config_RSPI0>_receive_interrupt as a callback routine.

[Syntax]

```
void r_<Config_RSPI0>_callback_receiveend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_callback_transmitend

This API function executes processing in response to transmit buffer empty interrupts or RSPI idle interrupts.

Remark This API function is called from r_<Config_RSPI0>_transmit_interrupt or r_<Config_RSPI0>_idle_interrupt as a callback routine.

[Syntax]

```
void    r_<Config_RSPI0>_callback_transmitend ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_RSPI0>_callback_error

This API function executes processing in response to error interrupts.

Remark This API function is called from r_<Config_RSPI0>_error_interrupt as a callback routine.

[Syntax]

```
void r_<Config_RSPI0>_callback_error ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Stopping communications after completing transmission and reception of the same number of bytes:

main.c

```
#include "r_smc_entry.h"
extern volatile uint32_t g_rspi0_tx_buf[4];
extern volatile uint32_t g_rspi0_rx_buf[4];
void main(void)
{
    /* Start the RSPI0 module operation */
    R_Config_RSPI0_Start();

    /* Send and receive RSPI0 data */
    R_Config_RSPI0_Send_Receive((uint32_t *)g_rspi0_tx_buf, 4U, (uint32_t *)g_rspi0_rx_buf);

    while (1U)
    {
        nop();
    }
}
```

Config_RSPI0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint32_t g_rspi0_tx_buf[4];
volatile uint32_t g_rspi0_rx_buf[4];
/* End user code. Do not edit comment generated here */

void R_Config_RSPI0_Create_UserInit(void)
{
    /* Start user code for user init. Do not edit comment generated here */
    g_rspi0_tx_buf[0] = 0x000000FF;
    g_rspi0_tx_buf[1] = 0x0000FF00;
    g_rspi0_tx_buf[2] = 0x00FF0000;
    g_rspi0_tx_buf[3] = 0xFF000000;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_RSPI0_callback_receiveend(void)
{
    /* Start user code for r_Config_RSPI0_callback_receiveend. Do not edit comment generated
here */
    /* Stop the RSPI0 module operation */
    R_Config_RSPI0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.37 Voltage Detection Circuit

The voltage detection circuit can be used to reset voltage monitoring 0 and reset/interrupt for voltage monitoring 1 and 2.

The Code Generator outputs the following API functions for the voltage detection circuit.

Table4.38 API Functions: [Voltage Detection Circuit]

API Function Name	Function
R_<Config_LVD1>_Create	Executes initialization processing that is required before controlling the voltage detection circuit.
R_<Config_LVD1>_Start	Starts monitoring of the voltage.
R_<Config_LVD1>_Stop	Stops monitoring of the voltage.
R_<Config_LVD1>_Create_UserInit	Executes user-specific initialization processing for the voltage detection circuit.
r_<Config_LVD1>_lvdn_interrupt	Executes processing in response to voltage monitoring interrupts.

R_<Config_LVD1>_Create

This API function executes initialization processing that is required before controlling the voltage detection circuit.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_LVD1>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LVD1>_Start

This API function starts monitoring of the voltage.

[Syntax]

```
void        R_<Config_LVD1>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LVD1>_Stop

This API function stops monitoring of the voltage.

[Syntax]

```
void        R_<Config_LVD1>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LVD1>_Create_UserInit

This API function executes user-specific initialization processing for the voltage detection circuit.

Remark This API function is called from R_<Config_LVD1>_Create as a callback routine.

[Syntax]

```
void R_<Config_LVD1>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_LVD1>_lvd*n*_interrupt

This API function executes processing in response to voltage monitoring interrupts.

[Syntax]

```
void        r_<Config_LVD1>_lvdn_interrupt ( void );
```

Remark *n* is a circuit number.

[Argument(s)]

None.

[Return value]

None.

Usage example

Generating a software reset in response to a voltage monitoring interrupt:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the LVD1 operation */
    R_Config_LVD1_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_LVD1_user.c

```
void r_Config_LVD1_lvd1_interrupt(void)
{
    /* Start user code for r_Config_LVD1_lvd1_interrupt. Do not edit comment generated here */
    /* Software reset */
    SYSTEM.PRCR.WORD = 0xA502;
    SYSTEM.SWRR = 0xA501;
    /* End user code. Do not edit comment generated here */
}
```

4.2.38 Watchdog Timer

The Code Generator outputs the following API functions for the watchdog timer (WDT or IWDT).

Table4.39 API Functions: [Watchdog Timer]

API Function Name	Function
R_<Config_WDT>_Create	Executes initialization processing that is required before controlling the watchdog timer (WDT or IWDT).
R_<Config_WDT>_Restart	Clears the counter in the watchdog timer, and then restarts counting by the counter.
R_<Config_WDT>_Create_UserInit	Executes user-specific initialization processing for the watchdog timer (WDT or IWDT).
r_<Config_WDT>_wuni_interrupt	Executes processing in response to maskable interrupts or non-maskable interrupts (WUNI).
r_<Config_WDT>_iwuni_interrupt	Executes processing in response to maskable interrupts or non-maskable interrupts (IWUNI).
r_<Config_WDT>_nmi_interrupt	Executes processing in response to non-maskable interrupts.

R_<Config_WDT>_Create

This API function executes initialization processing that is required before controlling the watchdog timer (WDT or IWDT).

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_WDT>_Create ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_WDT>_Restart

This API function clears the counter in the watchdog timer, and then restarts counting by the counter.

[Syntax]

```
void        R_<Config_WDT>_Restart ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_WDT>_Create_UserInit

This API function executes user-specific initialization processing for the watchdog timer (WDT or IWDT).

Remark This API function is called from R_<Config_WDT>_Create as a callback routine.

[Syntax]

```
void R_<Config_WDT>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_WDT>_wuni_interrupt

This API function executes processing in response to maskable interrupts or non-maskable interrupts (WUNI).

Remark This API function is called as the interrupt handler for maskable interrupts or non-maskable interrupts due to underflows or refresh errors of the down-counter.

[Syntax]

```
void        r_<Config_WDT>_wuni_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_WDT>_iwuni_interrupt

This API function executes processing in response to maskable interrupts or non-maskable interrupts (IWUNI).

Remark This API function is called as the interrupt handler for maskable interrupts or non-maskable interrupts due to underflows or refresh errors of the down-counter.

[Syntax]

```
void        r_<Config_WDT>_iwuni_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_WDT>_nmi_interrupt

This API function executes processing in response to non-maskable interrupts.

Remark This API function is called as the interrupt handler for non-maskable interrupts due to underflows or refresh errors of the down-counter.

[Syntax]

```
void    r_<Config_WDT>_nmi_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

Usage example

Refreshing the counter value on every loop of the main function and issuing a software reset in response to an underflow of the counter:

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    while (1U)
    {
        /* Restarts WDT module */
        R_Config_WDT_Restart();
    }
}
```

Config_WDT_user.c

```
void r_Config_WDT_wuni_interrupt(void)
{
    /* Start user code for r_Config_WDT_wuni_interrupt. Do not edit comment generated here */
    /* Software reset */
    SYSTEM.PRCR.WORD = 0xA502;
    SYSTEM.SWRR = 0xA501;
    /* End user code. Do not edit comment generated here */
}
```

4.2.39 Continuous Scan Mode DSAD

The Code Generator outputs the following API functions for the continuous scan mode DSAD.

Table4.40 API Functions: [Continuous Scan Mode DSAD]

API Function Name	Function
R_<Config_DSAD0>_Create	Executes initialization processing that is required before controlling the continuous scan mode DSAD.
R_<Config_DSAD0>_Start	Starts waiting trigger of A/D conversion.
R_<Config_DSAD0>_Stop	Sops A/D conversion.
R_<Config_DSAD0>_Set_SoftwareTrigger	Starts A/D conversion by software trigger.
R_<Config_DSAD0>_Get_ValueResult	Gets the result of conversion.
R_<Config_DSAD0>_Chm_Set_DisconnectDetection	Sets disconnect detection assist.
R_<Config_DSAD0>_Create_UserInit	Executes user-specific initialization processing for the continuous scan mode DSAD.
r_<Config_DSAD0>_adin_interrupt	Executes processing in response to conversion end interrupts.
r_<Config_DSAD0>_scanendn_interrupt	Executes processing in response to scan end interrupts.

R_<Config_DSAD0>_Create

This API function executes initialization processing that is required before controlling the continuous scan mode DSAD.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_DSAD0>_Create ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSAD0>_Start

This API function starts waiting trigger of A/D conversion.

[Syntax]

```
void        R_<Config_DSAD0>_Start ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSAD0>_Stop

This API function stops A/D conversion.

[Syntax]

```
void        R_<Config_DSAD0>_Stop ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSAD0>_Set_SoftwareTrigger

This API function starts A/D conversion by software trigger.

[Syntax]

```
void    R_<Config_DSAD0>_Set_SoftwareTrigger ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSAD0>_Get_ValueResult

This API function gets the result of conversion.

[Syntax]

```
void    R_<Config_DSAD0>_Get_ValueResult ( uint32_t * const buffer );
```

[Argument(s)]

I/O	Argument	Description
O	uint32_t * const *buffer*;	Pointer to the location where the acquired results of conversion are to be stored

[Return value]

None

R_<Config_DSAD0>_Ch*m*_Set_DisconnectDetection

This API function sets the disconnect detection assist.

[Syntax]

```
void    R_<Config_DSAD0>_Ch m _Set_DisconnectDetection ( bool pos, bool neg );
```

Remark1. *m* is channel numbers.

Remark2. Do not call this API during A/D conversion.

[Argument(s)]

I/O	Argument	Description
I	bool pos;	Enable to disconnect detection assist for positive input signal. True : Enable False : Disable
I	bool neg;	Enable to disconnect detection assist for negative input signal. True : Enable False : Disable

[Return value]

None

R_<Config_DSAD0>_Create_UserInit

This API function executes user-specific initialization processing for the continuous scan mode DSAD.

Remark This API function is called from R_<Config_DSAD0>_Create as a callback routine.

[Syntax]

```
void R_<Config_DSAD0>_Create_UserInit ( void );
```

[Argument(s)]

None

[Return value]

None

r_<Config_DSAD0>_adi*n*_interrupt

This API function executes processing in response to conversion end interrupts.

[Syntax]

```
void        r_<Config_DSAD0>_adin_interrupt ( void );
```

Remark *n* is unit numbers.

[Argument(s)]

None

[Return value]

None

r_<Config_DSAD0>_scanend*n*_interrupt

This API function executes processing in response to scan end interrupts.

[Syntax]

```
void        r_<Config_DSAD0>_scanend n_interrupt ( void );
```

Remark *n* is unit numbers.

[Argument(s)]

None

[Return value]

None

Usage example

Getting the result value when scan end interrupt occurs.

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the DSAD0 converter */
    R_Config_DSAD0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_DSAD0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint32_t g_dsad0_value;
/* End user code. Do not edit comment generated here */

void r_Config_DSAD0_scanend0_interrupt(void)
{
    /* Start user code for r_Config_DSAD0_scanend0_interrupt. Do not edit comment generated here */
    /* Get result from the DSAD0 converter */
    R_Config_DSAD0_Get_ValueResult((uint32_t *)&g_dsad0_value);
    /* End user code. Do not edit comment generated here */
}
```

4.2.40 Single Scan Mode DSAD

The Code Generator outputs the following API functions for the single scan mode DSAD.

Table4.41 API Functions [Single Scan Mode DSAD]

API Function Name	Function
R_<Config_DSAD0>_Create	Executes initialization processing that is required before controlling the single scan mode DSAD.
R_<Config_DSAD0>_Start	Starts waiting trigger of A/D conversion.
R_<Config_DSAD0>_Stop	Stops A/D conversion.
R_<Config_DSAD0>_Set_SoftwareTrigger	Starts A/D conversion by software trigger.
R_<Config_DSAD0>_Get_ValueResult	Gets the result of conversion.
R_<Config_DSAD0>_Chm_Set_DisconnectDetection	Sets disconnect detection assist.
R_<Config_DSAD0>_Create_UserInit	Executes user-specific initialization processing for the single scan mode DSAD.
r_<Config_DSAD0>_adin_interrupt	Executes processing in response to conversion end interrupts.
r_<Config_DSAD0>_scanendn_interrupt	Executes processing in response to scan end interrupts.

R_<Config_DSAD0>_Create

This API function executes initialization processing that is required before controlling the single scan mode DSAD.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_DSAD0>_Create ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSAD0>_Start

This API function starts waiting trigger of A/D conversion.

[Syntax]

```
void        R_<Config_DSAD0>_Start ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSAD0>_Stop

This API function stops A/D conversion.

[Syntax]

```
void        R_<Config_DSAD0>_Stop ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSAD0>_Set_SoftwareTrigger

This API function starts A/D conversion by software trigger.

[Syntax]

```
void    R_<Config_DSAD0>_Set_SoftwareTrigger ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSAD0>_Get_ValueResult

This API function gets the result of conversion.

[Syntax]

```
void    R_<Config_S12AD0>_Get_ValueResult ( uint32_t * const buffer );
```

[Argument(s)]

I/O	Argument	Description
O	uint32_t * const *buffer*;	Pointer to the location where the acquired results of conversion are to be stored

[Return value]

None

R_<Config_DSAD0>_Ch*m*_Set_DisconnectDetection

This API function sets the disconnect detection assist.

[Syntax]

```
void    R_<Config_DSAD0>_Ch m _Set_DisconnectDetection ( bool pos, bool neg );
```

Remark1. *m* is channel numbers.

Remark2. Do not call this API during A/D conversion.

[Argument(s)]

I/O	Argument	Description
I	bool pos;	Enable to disconnect detection assist for positive input signal. True : Enable False : Disable
I	bool neg;	Enable to disconnect detection assist for negative input signal. True : Enable False : Disable

[Return value]

None

R_<Config_DSAD0>_Create_UserInit

This API function executes user-specific initialization processing for the single scan mode DSAD.

Remark This API function is called from R_<Config_DSAD0>_Create as a callback routine.

[Syntax]

```
void R_<Config_DSAD0>_Create_UserInit ( void );
```

[Argument(s)]

None

[Return value]

None

r_<Config_DSAD0>_adi*n*_interrupt

This API function executes processing in response to conversion end interrupts.

[Syntax]

```
void        r_<Config_DSAD0>_adin_interrupt ( void );
```

Remark *n* is unit numbers.

[Argument(s)]

None

[Return value]

None

r_<Config_DSAD0>_scanend*n*_interrupt

This API function executes processing in response to scan end interrupts.

[Syntax]

```
void        r_<Config_DSAD0>_scanend n_interruptB ( void );
```

Remark *n* is unit numbers.

[Argument(s)]

None

[Return value]

None

Usage example

Getting the result value when scan end interrupt occurs.

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the DSAD0 converter */
    R_Config_DSAD0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_DSAD0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint32_t g_dsad0_value;
/* End user code. Do not edit comment generated here */

void r_Config_DSAD0_scanend0_interrupt(void)
{
    /* Start user code for r_Config_DSAD0_scanend0_interrupt. Do not edit comment generated here */
    /* Get result from the DSAD0 converter */
    R_Config_DSAD0_Get_ValueResult((uint32_t *)&g_dsad0_value);
    /* End user code. Do not edit comment generated here */
}
```

4.2.41 Delta-Sigma Modulator Interface

The Code Generator outputs the following API functions for the delta-sigma modulator interface.

Table4.42 API Functions [Delta-Sigma Modulator Interface]

API Function Name	Function
R_<Config_DSMIF0>_Create	Executes initialization processing that is required before controlling the delta-sigma modulator interface.
R_<Config_DSMIF0>_Start	Starts conversion
R_<Config_DSMIF0>_Stop	Stops conversion
R_<Config_DSMIF0>_Create_UserInit	Executes user-specific initialization processing for the delta-sigma modulator interface.
r_<Config_DSMIF0>_ocdin_interrupt	Executes processing in response to overcurrent detection interrupts.
r_<Config_DSMIF0>_scdin_interrupt	Executes processing in response to short-circuit detection interrupts.
r_<Config_DSMIF0>_sumein_interrupt	Executes processing in response to current sum error detection interrupts.

R_<Config_DSMIF0>_Create

This API function executes initialization processing that is required before controlling the delta-sigma modulator interface.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_DSMIF0>_Create ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSMIF0>_Start

This API function starts conversion.

[Syntax]

```
void        R_<Config_DSMIF0>_Start ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSMIF0>_Stop

This API function stops conversion.

[Syntax]

```
void        R_<Config_DSMIF0>_Stop ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_DSMIF0>_Create_UserInit

This API function executes user-specific initialization processing for the delta-sigma modulator interface.

Remark This API function is called from R_<Config_DSMIF0>_Create as a callback routine.

[Syntax]

```
void R_<Config_DSMIF0>_Create_UserInit ( void );
```

[Argument(s)]

None

[Return value]

None

r_<Config_DSMIF0>_ocdi*n*_interrupt

This API function executes processing in response to overcurrent detection interrupts.

[Syntax]

```
void        r_<Config_DSMIF0>_ocdin_interrupt ( void );
```

Remark *n* is unit numbers.

[Argument(s)]

None

[Return value]

None

r_<Config_DSMIF0>_scdi*n*_interrupt

This API function executes processing in response to short-circuit detection interrupts.

[Syntax]

```
void        r_<Config_DSMIF0>_scdin_interrupt ( void );
```

Remark *n* is unit numbers.

[Argument(s)]

None

[Return value]

None

r_<Config_DSMIF0>_sumei*n*_interrupt

This API function executes processing in response to current sum error detection interrupts.

[Syntax]

```
void        r_<Config_DSMIF0>_sumei n_interrupt ( void );
```

Remark *n* is unit numbers.

[Argument(s)]

None

[Return value]

None

Usage example

Stops conversion when over current detection.

main.c

```
#include "r_smc_entry.h"
void main(void)
{
    /* Start the DSMIF0 filltering */
    R_Config_DSMIF0_Start();

    while (1U)
    {
        nop();
    }
}
```

Config_DSMIF_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint16_t g_s12ad0_ch000_value;
/* End user code. Do not edit comment generated here */

void r_Config_DSMIF0_ocdi0_interrupt(void)
{
    /* Start user code for r_Config_DSMIF0_ocdi0_interrupt. Do not edit comment generated here
*/
    /* Stop the DSMIF0 convert */
    R_Config_DSMIF0_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.42 Analog Front End

The Code Generator outputs the following API functions for the analog front end.

Table4.43 API Functions [Delta-Sigma Modulator Interface]

API Function Name	Function
R_<Config_AFE>_Create	Executes initialization processing that is required before controlling the analog front end.
R_<Config_AFE>_Create_UserInit	Executes user-specific initialization processing for the analog front end.

R_<Config_AFE>_Create

This API function executes initialization processing that is required before controlling the analog front end.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_AFE>_Create ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_AFE>_Create_UserInit

This API function executes user-specific initialization processing for the analog front end.

Remark This API function is called from R_<Config_AFE>_Create as a callback routine.

[Syntax]

```
void R_<Config_AFE>_Create_UserInit ( void );
```

[Argument(s)]

None

[Return value]

None

4.2.43 Motor

The Code Generator outputs the following API functions for the motor.

Table4.44 API Functions [Motor]

API Function Name	Function
R_<Config_MTU3_MTU4>_Create	Executes initialization processing for MTU (complementary PWM mode) and S12AD (single scan mode) used for motor.
R_<Config_MTU3_MTU4>_StartTimerCount	Starts timer count
R_<Config_MTU3_MTU4>_StopTimerCount	Stops timer count
R_<Config_MTU3_MTU4>_StartTimerCtrl	Starts timer pulse output
R_<Config_MTU3_MTU4>_StopTimerCtrl	Stops timer pulse output
R_<Config_MTU3_MTU4>_UpdDuty	Updates the buffer of duty register
R_<Config_MTU3_MTU4>_StartAD	Starts A/D converter
R_<Config_MTU3_MTU4>_StopAD	Stops A/D converter
R_<Config_MTU3_MTU4>_AdcGetConvVal	Gets the A/D conversion result
R_<Config_MTU3_MTU4>_Create_UserInit	Executes user-specific initialization processing for motor configuration.
R_<Config_MTU3_MTU4>_CrestInterrupt	Executes processing in response to crest interrupt
r_<Config_MTU3_MTU4>_ad_interrupt	Executes processing in response to A/D conversion end interrupt

R_<Config_MTU3_MTU4>_Create

This API function executes initialization for MTU (complementary PWM mode) and S12AD (single scan mode) used for motor.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_MTU3_MTU4>_Create ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_MTU3_MTU4>_StartTimerCount

This API function starts timer count.

[Syntax]

```
void        R_<Config_MTU3_MTU4>_StartTimerCount ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_StopTimerCount

This API function stops timer count.

[Syntax]

```
void        R_<Config_MTU3_MTU4>_StopTimerCount ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_StartTimerCtrl

This API function starts timer pulse output.

[Syntax]

```
void        R_<Config_MTU3_MTU4>_StartTimerCtrl ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_StopTimerCtrl

This API function stops timer pulse output.

[Syntax]

```
void        R_<Config_MTU3_MTU4>_StopTimerCtrl ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_UpdDuty

This API function updates the buffer of duty register.

[Syntax]

3-Phase Brushless DC Motor

```
void    R_<Config_MTU3_MTU4>_UpdDuty (uint16_t duty_u, uint16_t duty_v, uint16_t duty_w);
```

2-Phase Stepping Motor

```
void    R_<Config_MTU3_MTU4>_UpdDuty (uint16_t duty_a, uint16_t duty_b);
```

[Argument(s)]

3-Phase Brushless DC Motor

I/O	Argument	Description
O	uint16_t duty_u	U phase duty register value
O	uint16_t duty_v	V phase duty register value
O	uint16_t duty_w	W phase duty register value

2-Phase Stepping Motor

I/O	Argument	Description
O	uint16_t duty_a	A phase duty register value
O	uint16_t duty_b	B phase duty register value

[Return value]

None.

R_<Config_MTU3_MTU4>_StartAD

This API function starts A/D converter.

[Syntax]

```
void        R_<Config_MTU3_MTU4>_StartAD ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_StopAD

This API function stops A/D converter.

[Syntax]

```
void        R_<Config_MTU3_MTU4>_StopAD ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_MTU3_MTU4>_AdcGetConvVal

This API function gets the A/D conversion result.

[Syntax]

3-Phase Brushless DC Motor

```
void    R_<Config_MTU3_MTU4>_AdcGetConvVal ( r_mtr_adc_tb *mtr_ad_data );
```

2-Phase Stepping Motor

```
void    R_<Config_MTU3_MTU4>_AdcGetConvVal ( r_mtr_adc_ts *mtr_ad_data );
```

[Argument(s)]

3-Phase Brushless DC Motor

I/O	Argument	Description
O	r_mtr_adc_tb *mtr_ad_data Structure Definition: typedef struct { uint16_t u2_iu_ad; uint16_t u2_iv_ad; uint16_t u2_iw_ad; uint16_t u2_vdc_ad; uint16_t u2_vphase_u_ad; uint16_t u2_vphase_v_ad; uint16_t u2_vphase_w_ad; } r_mtr_adc_tb;	Pointer to the location where the read value is to be stored

2-Phase Stepping Motor

I/O	Argument	Description
O	r_mtr_adc_ts *mtr_ad_data Structure Definition: typedef struct { uint16_t u2_ia_ad; uint16_t u2_ib_ad; uint16_t u2_vdc_ad; uint16_t u2_vphase_a_ad; uint16_t u2_vphase_b_ad; } r_mtr_adc_ts;	Pointer to the location where the read value is to be stored

[Return value]

None.

R_<Config_MTU3_MTU4>_Create_UserInit

This API function executes user-specific initialization processing for motor configuration.

Remark This API function is called from R_<Config_MTU3_MTU4>_Create as a callback routine.

[Syntax]

```
void R_<Config_MTU3_MTU4>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU3_MTU4>_CrestInterrupt

This API function executes processing in response to crest interrupt (TGIA3 or TGIA6).

Remark This API function is called as the interrupt handler for TGRA3/TGRA6 compare match interrupts, which occur when the current counter value (the value of the timer counter (TCNT)) matches a specified value (the value of the timer general register (TGRA3 or TGRA6)).

[Syntax]

```
void    r_<Config_MTU3_MTU4>_CrestInterrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

r_<Config_MTU3_MTU4>_ad_interrupt

This API function executes processing in response to A/D conversion end interrupt.

Remark Interrupt of one A/D converter unit that has smallest unit number in S12AD units selected on GUI is enabled.

[Syntax]

```
void r_<Config_MTU3_MTU4>_ad_interrupt ( void );
```

[Argument(s)]

None.

[Return value]

None.

4.2.44 LCD Controller

The Code Generator outputs the following API functions for the LCD Controller.

Table4.45 API Functions [LCD Controller]

API Function Name	Function
R_<Config_LCD >_Create	Executes initialization processing for LCD Controller.
R_<Config_LCD>_Start	Starts LCD display
R_<Config_LCD >_Stop	Stops LCD display
R_<Config_LCD>_Voltage_On	Enables voltage boost circuit or capacitor split circuit
R_<Config_LCD>_Voltage_Off	Disables voltage boost circuit or capacitor split circuit
R_<Config_LCD >_Create_UserInit	Executes user-specific initialization processing for LCD Controller configuration.

R_<Config_LCD>_Create

This API function executes initialization for LCD Controller.

Remark This API function is called from R_Systeminit before the main() function is executed.

[Syntax]

```
void R_<Config_LCD >_Create ( void );
```

[Argument(s)]

None

[Return value]

None

R_<Config_LCD>_Start

This API function starts LCD display.

[Syntax]

```
void        R_<Config_LCD>_Start ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LCD>_Stop

This API function stops LCD display.

[Syntax]

```
void        R_<Config_LCD>_Stop ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LCD>_Voltage_On

This API function enables voltage boost circuit or capacitor split circuit.

[Syntax]

```
void        R_<Config_LCD>_Voltage_On ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LCD>_Voltage_Off

This API function disables voltage boost circuit or capacitor split circuit.

[Syntax]

```
void        R_<Config_LCD>_Voltage_Off ( void );
```

[Argument(s)]

None.

[Return value]

None.

R_<Config_LCD>_Create_UserInit

This API function executes user-specific initialization processing for LCD Controller configurations.

Remark This API function is called from R_<Config_LCD>_Create as a callback routine.

[Syntax]

```
void R_<Config_LCD>_Create_UserInit ( void );
```

[Argument(s)]

None.

[Return value]

None.

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	July 01, 2018	–	First Edition issued
1.01	July 10, 2018	19	Modified remarks
		20	Modified section name
1.02	Nov. 05, 2019	19	Added initialization page
		480 – 508	Added new components - Continuous Scan Mode DSAD - Single Scan Mode DSAD - Delta-Sigma Modulator Interface
1.03	Jan. 20, 2020	11, 12, 18-19	Added file lists into output files table for components - General PWM Timer (GPT) - Low Power Consumption - Continuous Scan Mode DSAD - Single Scan Mode DSAD - Delta-Sigma Modulator Interface
		166, 174	Added API - Software counter clear
		253, 263	Added API - Set PGA operating condition setting
1.04	Jul. 20, 2020	200 – 232	Modified section structure - Separated API for RIIC and SCI simple I2C mode
		207	Modified remarks
1.05	Oct. 20, 2020	502 513	Added new API - Set disconnect detection assist setting
		527 – 529	Added new components - Analog front end
1.05	Oct. 20, 2020	530 - 542	Added new components - Motor
1.06	Feb.16, 2022	137 - 142	Splitted the Start and Stop API by U, V and W phases
		219 248	Modified the API name and parameter description
		299 300	Added new API - Start/Stop MTU channels' counters simultaneously
		455 – 460	Modified the data type description for transmit and receive buffer parameter
		476 477	Modified the data type description for transmit and receive buffer parameter
1.07	Aug.01, 2022	122 - 125	Updated DOC component - Added three parameters API description for the R_<Config_DOC>SetMode - Updated the value type with "uint32_t" support

Rev.	Date	Description	
		Page	Summary
1.07	Aug. 01, 2022	206	Added RSCI support information for the simple I2C mode
		375 - 418	- Added RSCI support information for the SCI/SCIF Asynchronous Mode component and SCI/SCIF Clock Synchronous Mode component - Updated the description of r_<Config_SCI0>_receive_interrupt API for the SCI/SCIF Asynchronous Mode component and SCI/SCIF Clock Synchronous Mode component
		437	Added RSCI support information for Smart Card Interface component
		452 – 472	- Added RSCI support information for the simple SPI mode - Added new API (Executes processing in response to receive error interrupts) - Updated the description of r_<Config_RSPI0>_receive_interrupt API for the SPI Clock Synchronous Mode component
		551 - 557	Added new component - LCD controller
1.08	Oct. 20, 2023	15, 334, 354, 355	Added new API - R_<Config_RTC>_Enable_Alarm_Interrupt - R_<Config_RTC>_Disable_Alarm_Interrupt
		151	Added remark for R_<Config_DMAC0>_Set_SoftwareTrigger
		173, 174	Added remark for R_<Config_GPT0>_Start and R_<Config_GPT0>_Stop
1.09	Oct 20, 2024	20	Added “WAIT_LOOP” information for infinite loop check
		112, 115	Added remark for R_<Config_DA>n_Start and R_<Config_DA>n_Sync_Start
		172, 173	Updated remark for R_<Config_GPT0>_Start and R_<Config_GPT0>_Stop
		308, 309	Added remark for R_<Config_POE>_Start and R_<Config_POE>_Stop
1.10	Apr 20, 2025	5	Added Cautions
		490	Added explanation for Voltage Detection Circuit

Smart Configurator User's Manual: RX API Reference

Publication Date: Rev.1.10 Apr 20, 2025

Published by: Renesas Electronics Corporation

Smart Configurator

Renesas Electronics Corporation

R20UT4360EJ0110