User's Manual

Smart Configurator

User's Manual: RISC-V MCU API Reference

RENESAS MCU
RISC-V MCU Family

Renesas Electronics
www.renesas.com

Rev.1.00 Jan 2024

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

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

www.renesas.com

Contact information

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

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

2. Processing at power-on

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

3. Input of signal during power-off state

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

4. Handling of unused pins

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

5. Clock signals

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

6. Voltage application waveform at input pin

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

7. Prohibition of access to reserved addresses

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

8. Differences between products

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

How to Use This Manual

Readers	The target readers of this manual are the application system engineers who use the Smart Configurator and need to understand its function.	
Purpose	The purpose of this manual is to explain the user for understanding and using the Smart Configurator 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

TABLE OF CONTENTS

1. GENERAL

This chapter gives an overview of the driver 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 Smart Configurator 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 Smart Configurator 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 Smart Configurator 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 Smart Configurator, user's source code within this comment is protected.

  [Comment for user source code descriptions]
  /* Start user code for xxxx. Do not edit comment generated here */
  /* End user code. Do not edit comment generated here */
  “xxxx” is changed for different user code:

  - “global” – user can add global variables and functions
  - “function” – user can add functions declaration in .h file
  - “user init” – user can add initializing code
  - Interrupt function name – user can add service routine code
  - “adding” – user can add functions in .c file
  - “include” – user can add including file in .c file
  - “pragma” – user can add pragma declaration in .c file

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

2. OUTPUT FILES

This chapter explains the file output by the Smart Configurator.

2.1 Description

The Smart Configurator outputs the following files.

Table 2-1 Output File List (1/9)

Component / Folder Name	File Name	API Function Name
General	{project name}.c	main
	r_smc_entry.h	—
	r_cg_systeminit.c	R_Systeminit
	r_cg_macrodriver.h	—
	r_cg_userdefine.h	—
	r_cg_interrupt_handlers.h	—
	r_cg_inthandler.c	—
	r_cg_vect_table.c	—
	r_cg_port.h	—
	r_cg_kr.h	—
	r_cg_wdt.h	—
	r_cg_icu.h	—
	r_cg_elc_common.c	R_ELC_Cancel_ModuleStop R_ELC_Enter_ModuleStop
	r_cg_elc.h	—
	r_cg_lvd.h	—
	r_cg_dtc_common.c	R_DTC_Cancel_ModuleStop R_DTC_Enter_ModuleStop
	r_cg_dtc_common_user.c	r_dtc_interrupt
	r_cg_dtc_common.h	—
	r_cg_dtc.h	—
	r_cg_tau_common.c	R_TAU*m*_Create R_TAU*m*_Cancel_ModuleStop R_TAU*m*_Enter_ModuleStop
	r_cg_tau_common.h	—
	r_cg_tau.h	—
	r_cg_itl_common.c	R_ITL_Create R_ITL_Start_Interrupt R_ITL_Stop Interrupt R_ITL_Cancel_ModuleStop R_ITL_Enter_ModuleStop
	r_cg_itl_common_user.c	r_itl_interrupt
	r_cg_itl_common.h	—
	r_cg_itl.h	—
	r_cg_rtc.h	—

Table 2-2 Output File List (2/9)

Component / Folder Name	File Name	API Function Name
General	r_cg_ad_common.c	R_ADC_Cancel_ModuleStop R_ADC_Enter_ModuleStop
	r_cg_ad_common.h	—
	r_cg_ad.h	—
	r_cg_da_common.c	R_DAC_Create R_DAC_Cancel_ModuleStop R_DAC_Enter_ModuleStop
	r_cg_da_common.h	—
	r_cg_da.h	—
	r_cg_comp_common.c	R_COMP_Create R_COMP_Cancel_ModuleStop R_COMP_Enter_ModuleStop
	r_cg_comp_common.h	—
	r_cg_comp.h	—
	r_cg_sau_common.c	R_SAU*m*_Create R_SAU*m*_Cancel_ModuleStop R_SAU*m*_Enter_ModuleStop R_SAU*m*_Set_SnoozeOn R_SAU*m*_Set_SnoozeOff
	r_cg_sau_common.h	—
	r_cg_sau.h	—
	r_cg_iica_common.c	R_IICA*n*_Cancel_ModuleStop R_IICA*n*_Enter_ModuleStop
	r_cg_iica_common.h	—
	r_cg_iica.h	—
	r_cg_uarta.h	—
	r_cg_remc_common.c	R_REMC_Cancel_ModuleStop R_REMC_Enter_ModuleStop
	r_cg_remc_common.h	—
	r_cg_remc.h	—
Low Voltage Detection	{Config_LVD*n*}.c	R_{Config_LVD*n*}_Create R_{Config_LVDn} _Start R_{Config_LVDn} _Stop
	{Config_LVD*n*}_user.c	R_{Config_LVD*n*}_Create_UserInit
	{Config_LVD*n*}.h	—
Interrupt Controller	{Config_ICU}.c	R_{Config_ICU}_Create R_{Config_ICU}_IRQn_Start R_{Config_ICU}_IRQn_Stop
	{Config_ICU}_user.c	R_{Config_ICU}_Create_UserInit r_{Config_ICU}_IRQn_interrupt
	{Config_ICU}.h	—

Table 2-3 Output File List (3/9)

Component / Folder Name	File Name	API Function Name
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	—
Event Link Controller	{Config_ELC}.c	R_{Config_ELC}_Create R_{Config_ELC}_Stop
	{Config_ELC}_user.c	R_{Config_ELC}_Create_UserInit
	{Config_ELC}.h	—
I/O Ports	{Config_PORT}.c	R_{Config_PORT}_Create R_{Config_PORT}_Set_Output_Level R_{Config_PORT}_Set_Output_Level_ELC
	{Config_PORT}_user.c	R_{Config_PORT}_Create_UserInit
	{Config_PORT}.h	—
Key Interrupt	{Config_KR}.c	R_{Config_KR}_Create R_{Config_KR} _Start R_{Config_KR} _Stop
	{Config_KR}_user.c	R_{Config_KR}_Create_UserInit r_{Config_KR}_interrupt
	{Config_KR}.h	—
Interval Timer (Timer Array Unit)	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Higher8bits_Start R_{Config_TAU*m_n*}_Higher8bits_Stop R_{Config_TAU*m_n*}_Lower8bits_Start R_{Config_TAU*m_n*}_Lower8bits_Stop
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_interrupt r_{Config_TAU*m_n*}_higher8bits_interrupt
	{Config_TAU*m_n*}.h	—
Square Wave Output (Timer Array Unit)	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Lower8bits_Start R_{Config_TAU*m_n*}_Lower8bits_Stop
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_interrupt
	{Config_TAU*m_n*}.h	—

Table 2-4 Output File List (4/9)

Component / Folder Name	File Name	API Function Name
External Event Counter (Timer Array Unit)	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Lower8bits_Start R_{Config_TAU*m_n*}_Lower8bits_Stop
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_ interrupt
	{Config_TAU*m_n*}.h	–
Divider Function (Timer Array Unit)	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_ interrupt
	{Config_TAU*m_n*}.h	–
Input Pulse Interval Measurement (Timer Array Unit)	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Get_PulseWidth
	{Config_TAU*m_n*}_user.c	r_{Config_TAU*m_n*}_ interrupt R_{Config_TAU*m_n*}_Create_UserInit
	{Config_TAU*m_n*}.h	–
Measurement of High-/Low-Level Width of Input Signal (Timer Array Unit)	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Get_PulseWidth
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_ interrupt
	{Config_TAU*m_n*}.h	–
Delay Counter (Timer Array Unit)	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Lower8bits_Start R_{Config_TAU*m_n*}_Lower8bits_Stop R_{Config_TAU*m_n*}_Set_SoftwareTriggerOn
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_ interrupt
	{Config_TAU*m_n*}.h	–

Table 2-5 Output File List (5/9)

Component / Folder Name	File Name	API Function Name
One-Shot Pulse Output (Timer Array Unit)	{Config_TAU*m*_*n*}.c	R_{Config_TAU*m*_*n*}_Create R_{Config_TAU*m*_*n*}_Start R_{Config_TAU*m*_*n*}_Stop R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn R_{Config_TAU*m*_*n*}_Set_Get_PulseWidth
	{Config_TAU*m*_*n*}_user.c	R_{Config_TAU*m*_*n*}_Create_UserInit r_{Config_TAU*m*_*n*}_channel*n*_interrupt r_{Config_TAU*m*_*n*}_channel*p*_interrupt
	{Config_TAU*m*_*n*}.h	–
PWM Output (Timer Array Unit)	{Config_TAU*m*_*n*}.c	R_{Config_TAU*m*_*n*}_Create R_{Config_TAU*m*_*n*}_Start R_{Config_TAU*m*_*n*}_Stop
	{Config_TAU*m*_*n*}_user.c	R_{Config_TAU*m*_*n*}_Create_UserInit r_{Config_TAU*m*_*n*}_channel*n*_interrupt r_{Config_TAU*m*_*n*}_channel*p*_interrupt
	{Config_TAU*m*_*n*}.h	–
Interval Timer (32-bit Interval Timer using 8-bit counter mode)	{Config_ITL*n*}.c	R_{Config_ITL*n*}_Create R_{Config_ITL*n*}_Start R_{Config_ITL*n*}_Stop R_{Config_ITL*n*}_Set_OperationMode R_{Config_ITL*n*}_Get_CaptureValue
	{Config_ITL*n*}_user.c	R_{Config_ITL*n*}_Create_UserInit r_{Config_ITL*n*}_Callback_Shared_Interrupt
	{Config_ITL*n*}.h	–
Interval Timer (32-bit Interval Timer using 16-bit counter mode)	{Config_ITL*n*_ITL*m*}.c	R_{Config_ITL*n*_ITL*m*}_Create R_{Config_ITL*n*_ITL*m*}_Start R_{Config_ITL*n*_ITL*m*}_Stop R_{Config_ITL*n*_ITL*m*}_Set_SoftwareTriggerOn R_{Config_ITL*n*_ITL*m*}_Set_OperationMode R_{Config_ITL*n*_ITL*m*}_Get_CaptureValue
	{Config_ITL*n*_ITL*m*}_user.c	R_{Config_ITL*n*_ITL*m*}_Create_UserInit r_{Config_ITLn_ITL*m*}_Callback_Shared_Interrupt
	{Config_ITL*n*_ITL*m*}.h	–

Table 2-6 Output File List (6/9)

Component / Folder Name	File Name	API Function Name
Interval Timer (32-bit Interval Timer using 32-bit counter mode)	{Config_ITL000_ITL001_ITL012_ITL013}.c	R_{Config_ITL000_ITL001_ITL012_ITL013}_Create R_{Config_ITL000_ITL001_ITL012_ITL013}_Start R_{Config_ITL000_ITL001_ITL012_ITL013}_Stop R_{Config_ITL000_ITL001_ITL012_ITL013}_Set_OperationMode
	{Config_ITL000_ITL001_ITL012_ITL013}_user.c	R_{Config_ITL000_ITL001_ITL012_ITL013}_Create_UserInit r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_Interrupt
	{Config_ITL000_ITL001_ITL012_ITL013}.h	—
Real-time Clock	{Config_RTC}.c	R_{Config_RTC}_Create R_{Config_RTC}_Start R_{Config_RTC}_Stop R_{Config_RTC}_Set_HourSystem R_{Config_RTC}_Set_CounterValue R_{Config_RTC}_Get_CounterValue R_{Config_RTC}_Set_ConstPeriodInterruptOn R_{Config_RTC}_Set_ConstPeriodInterruptOff R_{Config_RTC}_Set_AlarmOn R_{Config_RTC}_Set_AlarmOff R_{Config_RTC}_Set_AlarmValue R_{Config_RTC}_Get_AlarmValue R_{Config_RTC}_Set_RTC1HZOn R_{Config_RTC}_Set_RTC1HZOff
	{Config_RTC}_user.c	R_{Config_RTC}_Create_UserInit r_{Config_RTC}_interrupt r_{Config_RTC}_callback_constperiod r_{Config_RTC}_callback_alarm
	{Config_RTC}.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}_interrupt
	{Config_WDT}.h	—
Independant Watchdog Timer	{Config_IWDT}.c	R_{Config_IWDT}_Create R_{Config_IWDT} _Restart
	{Config_IWDT}_user.c	R_{Config_IWDT}_Create_UserInit r_{Config_IWDT}_interrupt
	{Config_IWDT}.h	—

Table 2-7 Output File List (7/9)

Component / Folder Name	File Name	API Function Name
Simplified SPI Communication	{Config_SPI*p*}.c	R_{Config_SPI*p*}_Create R_{Config_SPI*p*}_Start R_{Config_SPI*p*}_Stop R_{Config_SPI*p*}_Send R_{Config_SPI*p*}_Receive R_{Config_SPI*p*}_Send_Receive
	{Config_SPI*p*}_user.c	R_{Config_SPI*p*}_Create_UserInit r_{Config_SPI*p*}_interrupt r_{Config_SPI*p*}_callback_sendend r_{Config_SPI*p*}_callback_receiveend r_{Config_SPI*p*}_callback_error
	{Config_SPI*p*}.h	–
UART Communication (Serial array unit)	{Config_UART*q*}.c	R_{Config_UART*q*}_Create R_{Config_UART*q*}_Start R_{Config_UART*q*}_Stop R_{Config_UART*q*}_Send R_{Config_UART*q*}_Receive R_{Config_UART*q*}_Loopback_Enable R_{Config_UARTq}_Loopback_Disble
	{Config_UART*q*}_user.c	R_{Config_UART*q*}_Create_UserInit r_{Config_UART*q*}_interrupt_send r_{Config_UART*q*}_interrupt_receive r_{Config_UART*q*}_interrupt_error r_{Config_UART*q*}_callback_sendend r_{Config_UART*q*}_callback_receiveend r_{Config_UART*q*}_callback_error r_{Config_UART*q*}_callback_softwareoverrun
	{Config_UART*q*}.h	–
Simplified I2C Communication (Master mode) (Serial Array Unit)	{Config_IIC*r*}.c	R_{Config_IIC*r*}_Create R_{Config_IIC*r*}_StartCondition R_{Config_IIC*r*}_StopCondition R_{Config_IIC*r*}_Stop R_{Config_IIC*r*}_Master_Send R_{Config_IIC*r*}_Master_Receive
	{Config_IIC*r*}_user.c	R_{Config_IIC*r*}_Create_UserInit r_{Config_IIC*r*}_interrupt r_{Config_IIC*r*}_callback_master_sendend r_{Config_IIC*r*}_callback_master_receiveend r_{Config_IIC*r*}_callback_master_error
	{Config_IIC*r*}.h	–

Table 2-8 Output File List (8/9)

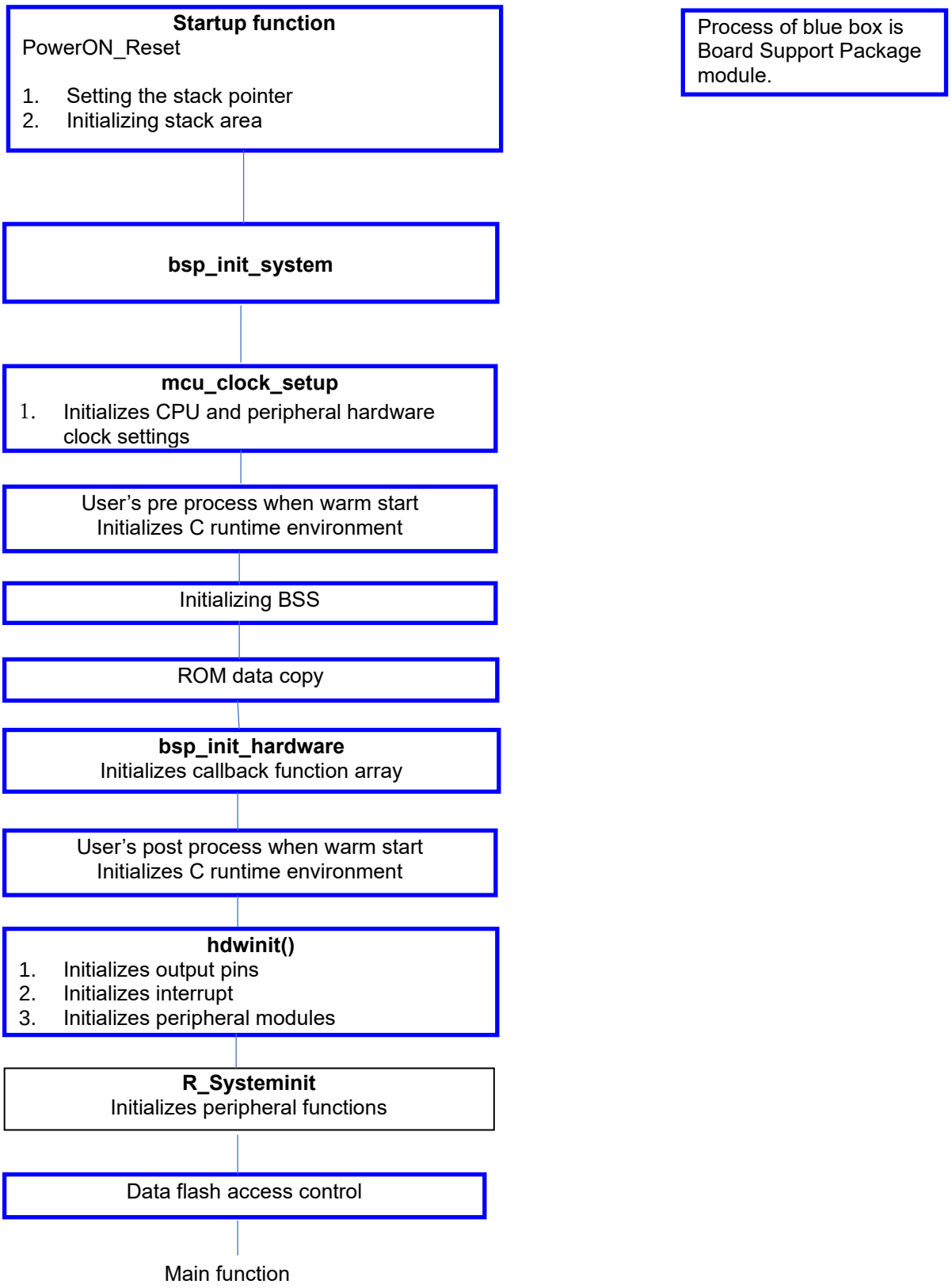
Component / Folder Name	File Name	API Function Name
I2C Bus Interface Communication (Master mode)	{Config_IICA*n*}.c	R_{Config_IICA*n*}_Create R_{Config_IICA*n*}_StopCondition R_{Config_IICA*n*}_Stop R_{Config_IICA*n*}_Master_Send R_{Config_IICA*n*}_Master_Receive
	{Config_IICA*n*}_user.c	R_{Config_IICA*n*}_Create_UserInit r_{Config_IICA*n*}_interrupt r_{Config_IICA*n*}_master_handler r_{Config_IICA*n*}_callback_master_sendend r_{Config_IICA*n*}_callback_master_receiveend r_{Config_IICA*n*}_callback_master_error
	{Config_IICA*n*}.h	–
I2C Bus Interface Communication (Slave mode)	{Config_IICA*n*}.c	R_{Config_IICA*n*}_Create R_{Config_IICA*n*}_Stop R_{Config_IICA*n*}_Slave_Send R_{Config_IICA*n*}_Slave_Receive R_{Config_IICA*n*}_Set_WakeupOn R_{Config_IICA*n*}_Set_WakeupOff
	{Config_IICA*n*}_user.c	R_{Config_IICA*n*}_Create_UserInit r_{Config_IICA*n*}_interrupt r_{Config_IICA*n*}_slave_handler r_{Config_IICA*n*}_callback_slave_sendend r_{Config_IICA*n*}_callback_slave_receiveend r_{Config_IICA*n*}_callback_slave_error r_{Config_IICA*n*}_callback_getstopcondition
	{Config_IICA*n*}.h	–
UART Communication (Serial Interface UARTA)	{Config_UARTA*n*}.c	R_{Config_UARTA*n*}_Create R_{Config_UARTA*n*}_Start R_{Config_UARTA*n*}_Stop R_{Config_UARTA*n*}_Send R_{Config_UARTA*n*}_Receive R_{Config_UARTA*n*}_Loopback_Enable R_{Config_UARTA*n*}_Loopback_Disable
	{Config_UARTA*n*}_user.c	R_{Config_UARTA*n*}_Create_UserInit R_{Config_UARTA*n*}_PollingEnd_UserCode r_{Config_UARTA*n*}_interrupt_send r_{Config_UARTA*n*}_interrupt_receive r_{Config_UARTA*n*}_interrupt_error r_{Config_UARTA*n*}_callback_sendend r_{Config_UARTA*n*}_callback_receiveend r_{Config_UARTA*n*}_callback_error
	{Config_UARTA*n*}.h	–

Table 2-9 Output File List (9/9)

Component / Folder Name	File Name	API Function Name
Remote Control Signal Receiver	{Config_REMC}.c	R_{Config_REMC}_Create R_{Config_REMC} _Start R_{Config_REMC} _Stop R_{Config_REMC}_Read
	{Config_REMC}_user.c	R_{Config_REMC}_Create_UserInit r_{Config_REMC}_interrupt r_{Config_REMC}_callback_receiveend r_{Config_REMC}_callback_comparematch r_{Config_REMC}_callback_receiveerror r_{Config_REMC}_callback_bufferfull r_{Config_REMC}_callback_header r_{Config_REMC}_callback_data0 r_{Config_REMC}_callback_data1 r_{Config_REMC}_callback_specialdata
	{Config_REMC}.h	—
A/D Convertor	{Config_ADC}.c	R_{Config_ADC}_Create R_{Config_ADC}_Start R_{Config_ADC}_Stop R_{Config_ADC}_Set_OperationOn R_{Config_ADC}_Set_OperationOff R_{Config_ADC}_Set_ADChannel R_{Config_ADC}_Set_SnoozeOn R_{Config_ADC}_Set_SnoozeOff R_{Config_ADC}_Set_TestChannel R_{Config_ADC}_Get_Result_10bit R_{Config_ADC}_Get_Result_8bit R_{Config_ADC}_Get_Result_12bit
	{Config_ADC}_user.c	R_{Config_ADC}_Create_UserInit r_{Config_ADC}_interrupt
	{Config_ADC}.h	—
D/A Converter	{Config_DAC*n*}.c	R_{Config_DAC*n*}_Create R_{Config_DAC*n*}_Start R_{Config_DAC*n*}_Stop R_{Config_DAC*n*}_Set_ConversionValue
	{Config_DAC*n*}_user.c	R_{Config_DAC*n*}_Create_UserInit
	{Config_DAC*n*}.h	—
Comparator	{Config_COMP*n*}.c	R_{Config_COMP*n*}_Create R_{Config_COMP*n*}_Start R_{Config_COMP*n*}_Stop
	{Config_COMP*n*}_user.c	R_{Config_COMP*n*}_Create_UserInit r_{Config_COMP*n*}_interrupt
	{Config_COMP*n*}.h	—

3. INITIALIZATION

This chapter describes the flow of initialization by the API functions of the Smart Configurator.



4. API FUNCTIONS

This chapter describes the API functions output that are output by the Smart Configurator.

4.1 Overview

The following are the naming conventions for the API functions output by the Smart Configurator.

- 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.

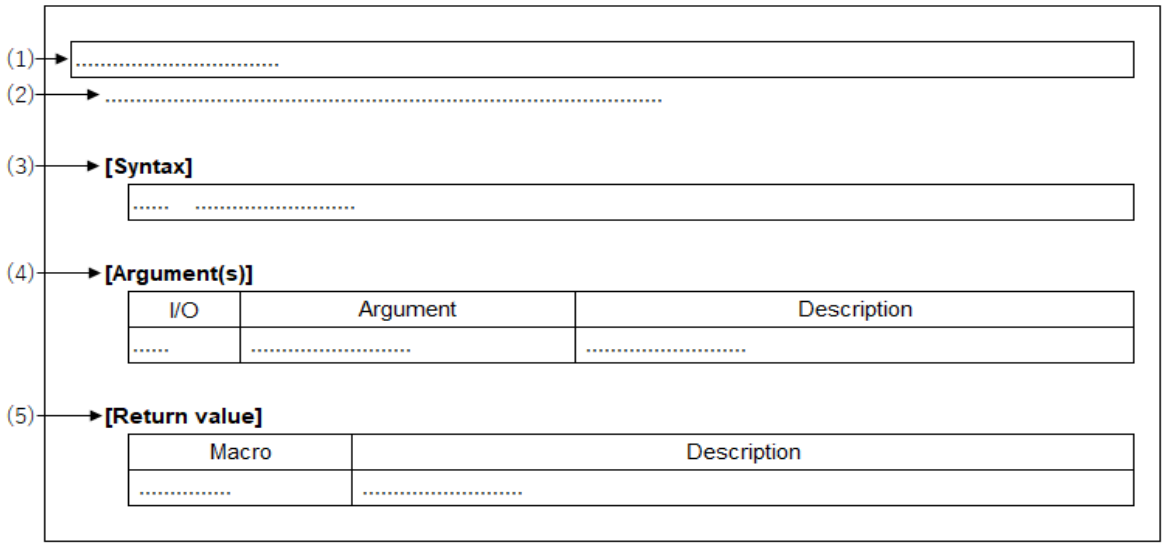
- Names of elements in enumeration specifiers “enum”
  These are in all-capital letters.

Remarks In the generated code by the Smart Configurator tool, 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.

4.2 Function Reference

This section describes the API functions output by the Smart Configurator, 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	Description
(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 General

Below is a list of API functions output by the Smart Configurator for common use.

Table 4-1 API Functions: (1/2)

API Function Name	Peripheral Name	Description
main	-	Main function.
R_Systeminit	-	Executes initialization processing that is required before controlling various peripheral modules.
R_DTC_Cancel_ModuleStop	Data Transfer Controller	Cancel the module-stop state for DTC.
R_DTC_Enter_ModuleStop		Enter the module-stop state for DTC.
R_ELC_Cancel_ModuleStop	Event Link Controller	Cancel the module-stop state for ELC.
R_ELC_Enter_ModuleStop		Enter the module-stop state for ELC.
R_TAU*m*_Create	Timer Array Unit	Executes initialization processing that is required before controlling TAU*m* (enables TAU*m* input clock supply and initializes TAU*m* module).
R_TAU*m*_Cancel_ModuleStop		Cancel the module-stop state for TAU*m*.
R_TAU*m*_Enter_ModuleStop		Enter the module-stop state for TAU*m*.
R_ITL_Create	32-Bit Interval Timer	Executes initialization processing that is required before controlling the 32-bits IT (enables input clock supply and initializes ITL*m* module).
R_ITL_Start_Interrupt		Starts 32-bit interval timer interrupt.
R_ITL_Stop Interrupt		Stops 32-bit interval timer interrupt.
R_ITL_Cancel_ModuleStop		Cancel the module-stop state for 32-bits IT.
R_ITL_Enter_ModuleStop		Enter the module-stop state for 32-bits IT.
R_ADC_Cancel_ModuleStop	A/D Converter	Cancel the module-stop state for AD converter.
R_ADC_Enter_ModuleStop		Enter the module-stop state for AD converter.
R_DAC_Create	D/A Converter	Executes initialization processing that is required before controlling the DAC module (enables input clock supply and initializes DAm module).
R_DAC_Cancel_ModuleStop		Cancel the module-stop state for DA converter.
R_DAC_Enter_ModuleStop		Enter the module-stop state for DA converter.
R_REMC_Cancel_ModuleStop	Remote Control Signal Receiver	Cancel the module-stop state for REMC.
R_REMC_Enter_ModuleStop		Enter the module-stop state for REMC.
R_COMP_Create	Comparator	Executes initialization processing that is required before controlling the COMP module (enables input clock supply and initializes COMP module).
R_COMP_Cancel_ModuleStop		Cancel the module-stop state for comparator.
R_COMP_Enter_ModuleStop		Enter the module-stop state for comparator.
R_SAU*m*_Create	Serial Array Unit	Executes initialization processing that is required before controlling SAU*m* (enables input clock supply and initializes SAU*m* module).
R_SAU*m*_Cancel_ModuleStop		Cancel the module-stop state for SAU*m*.
R_SAU*m*_Enter_ModuleStop		Enter the module-stop state for SAU*m*.
R_SAU*m*_Set_SnoozeOn		Enables SAU*m* wakeup function.
R_SAU*m*_Set_SnoozeOff		Disables SAU*m* wakeup function.

Table 4-2 API Functions: (2/2)

API Function Name	Peripheral Name	Description
R_UARTA_Create	Serial Interface UARTA	Executes initialization processing that is required before controlling UARTA0/UARTA1 (enables input clock supply and initializes module).
R_UARTA_Cancel_ModuleStop		Cancel the module-stop state for UARTA0/UARTA1.
R_IICA*n*_Cancel_ModuleStop	Serial Interface IICA	Cancel the module-stop state for IICA*n*.
R_IICA*n*_Enter_ModuleStop		Enter the module-stop state for IICA*n*.

main

This API function implements main function.

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_Systeminit

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

[Syntax]

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

[Argument(s)]

None.

[Return value]

None.

R_DTC_Cancel_ModuleStop

This API function Cancel the module-stop state for DTC.

[Syntax]

```
void    R_DTC_Cancel_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DTC_Enter_ModuleStop

This API function Enter the module-stop state for DTC.

[Syntax]

```
void    R_DTC_Enter_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

r_dtc_interrupt

This API function executes processing in response to Data Transfer Controller (DTC) interrupt.

Remark This API function is called as the interrupt handler for the CPU on a DTC activation interrupt or the CPU after a single data transfer or the CPU after a data transfer of a specified volume.

[Syntax]

```
void    r_dtc_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ELC_Cancel_ModuleStop

This API function Cancel the module-stop state for ELC.

[Syntax]

```
void    R_ELC_Cancel_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ELC_Enter_ModuleStop

This API function Enter the module-stop state for ELC.

[Syntax]

```
void    R_DTC_Enter_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_TAU*m*_Create

This API function executes initialization processing that is required before controlling TAU*m* (enables TAU*m* input clock supply and initializes TAU*m* module).

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

[Syntax]

```
void    R_TAUm_Create(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_TAU*m*_Cancel_ModuleStop

This API function Cancel the module-stop state for TAU*m*.

[Syntax]

```
void    R_TAUm_Cancel_ModuleStop(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_TAU*m*_Enter_ModuleStop

This API function Enter the module-stop state for TAU*m*.

[Syntax]

```
void    R_TAUm_Enter_ModuleStop(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_ITL_Create

This API function executes initialization processing that is required before controlling the 32-bits IT (enables input clock supply and initializes ITL*m* module).

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

[Syntax]

```
void    R_ITL_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ITL_Start_Interrupt

This API function starts 32-bit interval timer interrupt.

Remark The 32-bit interval timer interrupt is enabled by calling this API function. For this reason, to use 32-bit interval timer interrupt, please call this API function together with R_{Config_ITL000_ITL001_ITL012_ITL013}_Start or R_{Config_ITL*n*_ITL*m*}_Start or R_{Config_ITL*n*}_Start.

[Syntax]

```
void    R_ITL_Start_Interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ITL_Stop_Interrupt

This API function stops 32-bit interval timer interrupt.

[Syntax]

```
void    R_ITL_Stop_Interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ITL_Cancel_ModuleStop

This API function Cancel the module-stop state for 32-bits IT.

[Syntax]

```
void    R_ITL_Cancel_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ITL_Enter_ModuleStop

This API function Enter the module-stop state for 32-bits IT.

[Syntax]

```
void    R_ITL_Enter_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

r_itl_interrupt

This API function executes processing in response to 32-bit interval timer interrupt.

Remark This API function is called as the interrupt handler for compare match interrupt, which occur when the counter value in any of channels 0 to 3 matches the compare value.

[Syntax]

```
void    r_itl_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ADC_Cancel_ModuleStop

This API function Cancel the module-stop state for AD converter.

[Syntax]

```
void    R_ADC_Cancel_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_ADC_Enter_ModuleStop

This API function Enter the module-stop state for AD converter.

[Syntax]

```
void    R_ADC_Enter_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DAC_Create

This API function executes initialization processing that is required before controlling the DAC module (enables input clock supply and initializes DA*m* module).

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

[Syntax]

```
void    R_DAC_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DAC_Cancel_ModuleStop

This API function Cancel the module-stop state for DA converter.

[Syntax]

```
void    R_DAC_Cancel_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_DAC_Enter_ModuleStop

This API function Enter the module-stop state for DA converter.

[Syntax]

```
void    R_DAC_Enter_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_COMP_Create

This API function executes initialization processing that is required before controlling the comparator module (enables input clock supply and initializes COMP*m* module).

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

[Syntax]

```
void    R_COMP_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_COMP_Cancel_ModuleStop

This API function Cancel the module-stop state for comparator.

[Syntax]

```
void    R_COMP_Cancel_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_COMP_Enter_ModuleStop

This API function Enter the module-stop state for comparator.

[Syntax]

```
void    R_COMP_Enter_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_SAU*m*_Create

This API function executes initialization processing that is required before controlling SAU*m* (enables input clock supply and initializes SAU*m* module).

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

[Syntax]

```
void    R_SAUm_Create(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_SAU*m*_Cancel_ModuleStop

This API function Cancel the module-stop state for SAU*m*.

[Syntax]

```
void    R_SAUm_Cancel_ModuleStop(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_SAU*m*_Enter_ModuleStop

This API function Enter the module-stop state for SAU*m*.

[Syntax]

```
void    R_SAUm_Enter_ModuleStop(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_SAU*m*_Set_SnoozeOn

This API function enables SAU*m* wakeup function.

[Syntax]

```
void    R_SAUm_Set_SnoozeOn(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_SAU*m*_Set_SnoozeOff

This API function disables SAU*m* wakeup function.

[Syntax]

```
void    R_SAUm_Set_SnoozeOff(void);
```

Remark *m* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_UARTA_Create

This API function executes initialization processing that is required before controlling UARTA0/UARTA1 (enables input clock supply and initializes module).

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

[Syntax]

```
void    R_UARTA_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_UARTA_Cancel_ModuleStop

This API function Cancel the module-stop state for UARTA0/UARTA1.

[Syntax]

```
void    R_UARTA_Cancel_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_UARTA_Enter_ModuleStop

This API function Enter the module-stop state for UARTA0/UARTA1.

[Syntax]

```
void    R_UARTA_Enter_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_IICA*n*_Cancel_ModuleStop

This API function Cancel the module-stop state for IICA*n*.

[Syntax]

```
void    R_IICAn_Cancel_ModuleStop(void);
```

Remark *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_IICA*n*_Enter_ModuleStop

This API function Enter the module-stop state for IICA*n*.

[Syntax]

```
void    R_IICAn_Enter_ModuleStop(void);
```

Remark *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

R_REMC_Cancel_ModuleStop

This API function Cancel the module-stop state for REMC.

[Syntax]

```
void    R_REMC_Cancel_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_REMC_Enter_ModuleStop

This API function Enter the module-stop state for REMC.

[Syntax]

```
void    R_REMC_Enter_ModuleStop(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using R_*Xxxx*_Cancel_ModuleStop(), R_*Xxxx*_Enter_ModuleStop(), R_*Xxxx*_Start_Interrupt(), R_*Xxxx*_Stop_Interrupt():
(*Xxxx* is peripheral name which user want to use, the following sample code takes 32-bit Interval Timer(TML32) as an example)

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    // To enable 32-bit interval timer interrupt which is shared among TML32 each channels
    R_ITL_Start_Interrupt();

    R_Config_ITL000_Start();
    R_Config_ITL001_Start();
    R_Config_ITL000_Stop();
    R_Config_ITL001_Stop();

    // To Disable 32-bit interval timer interrupt which is shared among TML32 each channels
    R_ITL_Stop_Interrupt();

    // When ITL is stopped, to reduce the power consumption and noise
    R_ITL_Enter_ModuleStop();

    R_ITL_Create();
    R_Config_ITL000_Start();
    R_Config_ITL000_Stop();

    return 0;
}
```

4.2.2 Low Voltage Detection

Below is a list of API functions output by the Smart Configurator for low voltage detection use.

Table 4-3 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_LVD*n*}_Create	Voltage Detector	Executes initialization processing that is required before controlling the voltage detector module.
R_{Config_LVDn}_Start		Starts the voltage detector operation.
R_{Config_LVDn}_Stop		Stops the voltage detector operation.
R_{Config_LVD*n*}_Create_UserInit		Executes user-specific initialization processing for the voltage detector.

R_{Config_LVD*n*}_Create

This API function executes initialization processing that is required before controlling the voltage detector module.

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

[Syntax]

```
void    R_{Config_LVDn}_Create(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_LVDn}_Start

This API function starts the voltage detector operation.

[Syntax]

```
void    R_{Config_LVDn}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_LVDn}_Stop

This API function stops the voltage detector operation.

[Syntax]

```
void    R_{Config_LVDn}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_LVD*n*}_Create_UserInit

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

Remark This API functions is called from R_{Config_LVD*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_LVDn}_Create_UserInit(void);
```

Remark *n* is the unit number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for LVD operating in interrupt mode:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    R_Config_LVD1_Start();
    while(1);

    return 0;

}
```

4.2.3 Interrupt Controller

Below is a list of API functions output by the Smart Configurator for interrupt controller use.

Table 4-4 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_ICU}_Create	Interrupt function	Executes initialization processing that is required before controlling interrupt detection selection module.
R_{Config_ICU}_IRQn_Start		Clears IRQn interrupt flag and enables interrupt.
R_{Config_ICU}_IRQn_Stop		Disables IRQn interrupt and clears interrupt flag.
R_{Config_ICU}_Create_UserInit		Executes user-specific initialization processing for the interrupt detection selection module.
r_{Config_ICU}_IRQn_interrupt		Executes processing in response to IRQn interrupt.

R_{Config_ICU}_Create

This API function executes initialization processing that is required before controlling the interrupt detection selection module.

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}_IRQn_Start

This API function clears IRQn interrupt flag and enables interrupt.

[Syntax]

```
void    R_{Config_ICU}_IRQn_Start(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ICU}_IRQn_Stop

This API function disables IRQn interrupt and clears interrupt flag.

[Syntax]

```
void    R_{Config_ICU}_IRQn_Stop(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ICU}_Create_UserInit

This API function executes user-specific initialization processing for the interrupt detection selection module.

Remark This API functions 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}_IRQn_interrupt

This API function executes processing in response to IRQn interrupt.

[Syntax]

```
void    r_{Config_ICU}_IRQn_interrupt(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for setting a flag when detecting IRQ0 valid edge input:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t irq0_int_flag;

int main(void);

int main(void)
{

      irq0_int_flag = 0U;
      R_Config_ICU_IRQ0_Start ();
      while(intp0_int_flag != 1U);
      R_Config_ ICU_IRQ0_Stop ();
}
```

Config_ICU_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t irq0_int_flag = 0U;
/* End user code. Do not edit comment generated here */

static void __near r_Config_ICU_irq0_interrupt(void)
{
      /* Start user code for r_Config_ICU_irq0_interrupt. Do not edit comment generated here */
      /* Set the flag */
      irq0_int_flag = 1U;
      /* End user code. Do not edit comment generated here */
}
```

4.2.4 Data Transfer Controller

Below is a list of API functions output by the Smart Configurator for data transfer controller use.

Table 4-5 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_DTC}_Create	Data Transfer Controller	Executes initialization processing that is required before controlling the DTC module.
R_{Config_DTC}_Start		Starts DTC module operation.
R_{Config_DTC}_Stop		Stops DTC module operation.
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 DTC module.

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 starts DTC module operation

[Syntax]

```
void    R_{Config_DTC}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_DTC}_Stop

This API function stops DTC module operation

[Syntax]

```
void    R_{Config_DTC}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_DTC}_Create_UserInit

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

Remark This API functions 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.

Usage example

This is an example for using DTC data transfer in response to fixed-cycle signal of real-time clock/alarm match detection:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    R_DTC_Start();
    R_Config_RTC_Start();

    R_Config_RTC_Stop();
    R_DTC_Stop();

    return 0;
}
```

4.2.5 Event Link Controller

Below is a list of API functions output by the Smart Configurator for event link controller (ELC) use.

Table 4-6 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_ELC}_Create	Event Link Controller	Executes initialization processing that is required before controlling the ELC module.
R_{Config_ELC}_GenerateSoftwareEvent1		Triggers a software event.
R_{Config_ELC}_GenerateSoftwareEvent2		Triggers a software event.
R_{Config_ELC}_Start		Enables all ELC event links.
R_{Config_ELC}_Stop		Disables all ELC event links.
R_{Config_ELC}_Create_UserInit		Executes user-specific initialization processing for the ELC.

R_{Config_ELC}_Create

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

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}_GenerateSoftwareEvent1

This API function triggers a software event.

[Syntax]

```
void    R_{Config_ELC}_GenerateSoftwareEvent1( void );
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ELC}_GenerateSoftwareEvent2

This API function triggers a software event.

[Syntax]

```
void    R_{Config_ELC}_GenerateSoftwareEvent2( void );
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ELC}_Start

This API function enables all ELC event links.

[Syntax]

```
void    R_{Config_ELC}_Start( void );
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ELC}_Stop

This API function stops all ELC event links.

[Syntax]

```
void    R_{Config_ELC}_Stop( void );
```

[Argument(s)]

None

[Return value]

None.

R_{Config_ELC}_Create_UserInit

This API function executes user-specific initialization processing for ELC.

Remark This API functions 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.

Usage example

This is an example for ELC is used to start ELC event generation:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    R_Config_ELC_Start();
    R_Config_ELC_GenerateSoftwareEvent1();

    while(1);

    return 0;
}
```

4.2.6 I/O ports

Below is a list of API functions output by the Smart Configurator for I/O ports use.

Table 4-7 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_PORT}_Create	I/O Port	Executes initialization processing that is required before controlling the I/O ports.
R_{Config_PORT}_Set_Output_Level		Set pin output level.
R_{Config_PORT}_Set_Output_Level_ELC		Set pin output level when an ELC_PORTx signal occurs.
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}_Set_Output_Level

This API function set specified pin output level.

[Syntax]

```
MD_STATUS      R_{Config_PORT}_Set_Output_Level(char Pmn[], e_output_level_t outputlevel);
```

[Argument(s)]

I/O	Argument(s)	Description
I	char Pmn[]	Specify the pin you want to set, ex: ‘p101’.
I	e_output_level_t outputlevel	Define high/low output level

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_PORT}_Set_Output_Level_ELC

This API function specifies the output level on a port pin is read when the pin is in output mode.

[Syntax]

```
void    R_{Config_PORT}_Set_Output_Level_ELC(char Pmn[], e_output_level_t outputlevel);
```

[Argument(s)]

I/O	Argument(s)	Description
I	char Pmn[]	Specify the pin you want to set, ex: ‘p101’.
I	e_output_level_t outputlevel	Define high/low output level

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_Config_PORT_Create_UserInit

This API function executes user-specific initialization processing for the port I/O.

Remark This API functions is called as the R_{Config_PORT}_Create callback routine.

[Syntax]

```
void    R_{Config_PORT}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for setting the output level on a port pin is read when the pin is in output mode:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    char Pmn[] = "P101";

    R_Config_PORT_Set_Output_Level(Pmn, LOW);


    return 0;
}
```

4.2.7 Key Interrupt

Below is a list of API functions output by the Smart Configurator for key Interrupt use.

Table 4-8 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_KR}_Create	Key Interrupt	Executes initialization processing that is required before controlling the key interrupt module.
R_{Config_KR}_Start		Clears key interrupt flag and enables interrupt.
R_{Config_KR}_Stop		Disables key interrupt and clears interrupt flag.
R_{Config_KR}_Create_UserInit		Executes user-specific initialization processing for the key interrupt.
r_{Config_KR}_interrupt		Executes processing in response to key interrupt.

R_{Config_KR}_Create

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

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

[Syntax]

```
void    R_{Config_KR}_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_KR}_Start

This API function clears key interrupt flag and enables interrupt.

[Syntax]

```
void    R_{Config_KR}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_KR}_Stop

This API function disables key interrupt and clears interrupt flag.

[Syntax]

```
void    R_{Config_KR}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_KR}_Create_UserInit

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

Remark This API functions is called from R_{Config_KR}_Create as a callback routine.

[Syntax]

```
void    R_{Config_KR}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_KR}_interrupt

This API function executes processing in response to key interrupt.

[Syntax]

```
void    r_{Config_KR}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for setting a flag when detecting key interrupt:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t kr_int_flag;

int main(void);

int main(void)
{

      kr_int_flag = 0U;
      R_Config_KR_Start ();
      while(kr_int_flag != 1U);
      R_Config_ KR_Stop ();

      return 0;
}
```

Config_KR_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t kr_int_flag = 0U
/* End user code. Do not edit comment generated here */

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

4.2.8 Interval Timer (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for interval timer (for timer array unit) use.

Table 4-9 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in interval timer mode.
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Higher8bits_Start		Starts the TAU*m* channel*n* higher 8 bits counter.
R_{Config_TAU*m*_*n*}_Higher8bits_Stop		Stops the TAU*m* channel*n* higher 8 bits counter.
R_{Config_TAU*m*_*n*}_Lower8bits_Start		Starts the TAU*m* channel*n* lower 8 bits counter.
R_{Config_TAU*m*_*n*}_Lower8bits_Stop		Stops the TAU*m* channel*n* lower 8 bits counter.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channel*n*.
r_{Config_TAU*m*_*n*}_interrupt		Executes processing in response to timer channel*n* count end interrupt.
r_{Config_TAU*m*_*n*}_higher8bits_interrupt		Executes processing in response to timer channel*n* count end interrupt at higher 8-bit timer operation.

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channel*n* module in interval timer mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channel*n* counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channel*n* counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Higher8bits_Start

Starts the TAU*m* channel*n* higher 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Higher8bits_Start(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Higher8bits_Stop

Stops the TAU*m* channel*n* higher 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Higher8bits_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Lower8bits_Start

Starts the TAU*m* channel*n* lower 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Lower8bits_Start(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Lower8bits_Stop

Stops the TAU*m* channel*n* lower 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Lower8bits_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channel*n*.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_interrupt

This API function executes processing in response to timer channel*n* count end interrupt.

Remark This API function is called as the interrupt handler for count end interrupt, which occur when the current counter value (TCR*mn*) reaches 0000H.

[Syntax]

```
void    r_{Config_TAUm_n}_interrupt(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_higher8bits_interrupt

This API function executes processing in response to timer channel*n* count end interrupt at higher 8-bit timer operation.

Remark This API function is called as the interrupt handler for count end interrupt, which occur when the current counter (TCR*mn*) higher 8-bit value reaches 00H.

[Syntax]

```
void    r_{Config_TAUm_n}_higher8bits_interrupt(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TAU channel 0 counting interval timer, channel 3 counting as high 8-bit interval timer and channel 1 counting as low 8-bit interval timer for a user-defined period:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;
extern uint8_t ch1_run_count;
extern uint8_t ch3_run_count;

int main(void);

int main(void)
{
    R_Config_TAU0_0_Start();
    while (ch0_run_count < 20);
    R_Config_TAU0_0_Stop();

    R_Config_TAU0_1_Lower8bits_Start();
    while (ch1_run_count < 20);
    R_Config_TAU0_1_Lower8bits_Stop();

    R_Config_TAU0_3_Higher8bits_Start();
    while (ch3_run_count < 20);
    R_Config_TAU0_3_Higher8bits_Stop();

    return 0;
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_interrupt. Do not edit comment generated here */
    ch1_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_3_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch3_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_3 _interrupt (void)
{
    /* Start user code for r_Config_TAU0_3 _interrupt. Do not edit comment generated here */
    ch3_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.9 Square Wave Output (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for square wave output use.

Table 4-10 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in square wave output mode.
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Lower8bits_Start		Starts the TAU*m* channel*n* lower 8 bits counter.
R_{Config_TAU*m*_*n*}_Lower8bits_Stop		Stops the TAU*m* channel*n* lower 8 bits counter.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channel*n*.
r_{Config_TAU*m*_*n*}_interrupt		Executes processing in response to timer channel*n* measurement end interrupt.

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channeln module in square wave output mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Lower8bits_Start

This API function starts the TAU*m* channeln lower 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Lower8bits_Start(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Lower8bits_Stop

This API function stops the TAU*m* channeln lower 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Lower8bits_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channeln.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_interrupt

This API function executes processing in response to timer channelmn count end interrupt.

Remark This API function is called as the interrupt handler for measurement end interrupt, which occur when the current counter value (TCRmn) reaches 0000H.

[Syntax]

```
void    r_{Config_TAUm_n}_interrupt(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TAU channel 0 counter and channel 1 lower 8-bit counter to output square wave for a user-defined period:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;
extern uint8_t ch1_run_count;

int main(void);

int main(void)
{
    R_Config_TAU0_0_Start();
    while (ch0_run_count < 20);
    R_Config_TAU0_0_Stop();

    R_Config_TAU0_1_Lower8bits_Start();
    while (ch1_run_count < 20);
    R_Config_TAU0_1_Lower8bits_Stop();

    return 0;
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count++;
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_interrupt. Do not edit comment generated here */
    ch1_run_count++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.10 External Event Counter (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for external event counter use.

Table 4-11 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in external event counter mode.
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Lower8bits_Start		Starts the TAU*m* channel*n* lower 8 bits counter.
R_{Config_TAU*m*_*n*}_Lower8bits_Stop		Stops the TAU*m* channel*n* lower 8 bits counter.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channel*n*.
r_{Config_TAU*m*_*n*}_interrupt		Executes processing in response to end of timer channel*n* count end interrupt.

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channeln module in external event counter mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Lower8bits_Start

This API function starts the TAU*m* channeln lower 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Lower8bits_Start(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Lower8bits_Stop

This API function stops the TAU*m* channeln lower 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Lower8bits_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channeln.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_interrupt

This API function executes processing in response to end of timer channelmn count end interrupt.

Remark This API function is called as the interrupt handler for count end interrupt, which occur when the current counter value (TCRmn) reaches 0000H.

[Syntax]

```
void    r_{Config_TAUm_n}_interrupt(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TAU channel 0 counting as external event counter and channel 1 counting as 8-bit external event counter for a user-defined period:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;
extern uint8_t ch1_run_count;

int main(void);

int main(void)
{
    R_Config_TAU0_0_Start();
    while (ch0_run_count < 20);
    R_Config_TAU0_0_Stop();

    R_Config_TAU0_1_Lower8bits_Start();
    while (ch1_run_count < 20);
    R_Config_TAU0_1_Lower8bits_Stop();

    return 0;
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_interrupt. Do not edit comment generated here */
    ch1_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.11 Divider Function (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for divider function use.

Table 4-12 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in divider function mode.
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channel.
r_{Config_TAU*m*_*n*}_interrupt		Executes processing in response to end of timer channel*n* count end interrupt.

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channeln module in divider function mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channeln.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_interrupt

This API function executes processing in response to end of timer channelmn count end interrupt.

Remark This API function is called as the interrupt handler for count end interrupt, which occur when the current counter value (TCRmn) reaches 0000H.

[Syntax]

```
void    r_{Config_TAUm_n}_interrupt(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TAU channel 0 counting as divider mode for a user-defined period:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count

int main(void);

int main(void)
{
    R_Config_TAU0_0_Start();
    while( ch0_run_count < 20);
    R_Config_TAU0_0_Stop();

    return 0;
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.12 Input Pulse Interval Measurement (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for input pulse interval measurement use.

Table 4-13 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in input pulse interval measurement mode
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Get_PulseWidth		Measures TAU*m* channel*n* input pulse width.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channel*n*.
r_{Config_TAU*m*_*n*}_interrupt		Executes processing in response to timer channel*n* capture interrupt.

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channeln module in input pulse interval measurement mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Get_PulseWidth

Measures TAU*m* channel n input pulse width.

[Syntax]

```
void    R_{Config_TAUm_n}_Get_PulseWidth(uint32_t * const width);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
O	uint32_t * const width;	the address where to write the input pulse width

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channeln.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_interrupt

This API function executes processing in response to timer channelmn capture interrupt.

Remark This API function is called as the interrupt handler for capture interrupts which occur when the valid capture edge is detected and the current counter value (TCRmn) is transferred to timer data register mn (TDRmn).

[Syntax]

```
void    r_{Config_TAUm_n}_interrupt(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting TAU channel 0 input interval width:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t tau_interrupt_flag;
uint32_t width;

int main(void);

int main(void)
{
    tau_interrupt_flag = 0;
    R_Config_TAU0_0_Start();
    while (tau_interrupt_flag == 0);
    R_Config_TAU0_0_Stop();
    R_Config_TAU0_0_Get_PulseWidth(&width);

    return 0;
}
```

Config_TAU0_0_user.c

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

static void __near r_Config_TAU0_0_interrupt (void)
{
    ...
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    tau_interrupt_flag = 1U;       // Set the flag
    /* End user code. Do not edit comment generated here */
}
```

4.2.13 Measurement of High-/Low-Level Width of Input Signal (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for input pulse high-/low-level width measurement use.

Table 4-14 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in input pulse high-/low-level width measurement mode.
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Get_PulseWidth		Measures TAU*m* channel*n* input pulse width.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channel*n*.
r_{Config_TAU*m*_*n*}_interrupt		Executes processing in response to timer channel*n* capture interrupt.

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channeln module in input pulse high-/low-level width measurement mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Get_PulseWidth

This API function measures TAU*m* channeln input pulse width.

[Syntax]

```
void    R_{Config_TAUm_n}_Get_PulseWidth(uint32_t * const width);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
O	uint32_t * const width;	The address where to write the input pulse width

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channeln.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_interrupt

This API function executes processing in response to timer channelmn capture interrupt.

Remark This API function is called as the interrupt handler for capture interrupts, which occur when the valid capture edge is detected, and the current counter value (TCRmn) is transferred to timer data register mn (TDRmn).

[Syntax]

```
void    r_{Config_TAUm_n}_interrupt(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting TAU channel 0 input low-level width:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t tau_interrupt_flag;
uint32_t width;

int main(void);

int main(void)
{
    tau_interrupt_flag = 0;
    R_Config_TAU0_0_Start();
    while( tau_interrupt_flag == 0);
    R_Config_TAU0_0_Stop();
    R_Config_TAU0_0_Get_PulseWidth(&width);

    return 0;
}
```

Config_TAU0_0_user.c

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

static void __near r_Config_TAU0_0_interrupt (void)
{
    ...
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    tau_interrupt_flag = 1U;       // Set the flag
    /* End user code. Do not edit comment generated here */
}
```

4.2.14 Delay Counter (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for delay counter use.

Table 4-15 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in delay counter mode.
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Lower8bits_Start		Starts the TAU*m* channel*n* lower 8 bits counter.
R_{Config_TAU*m*_*n*}_Lower8bits_Stop		Stops the TAU*m* channel*n* lower 8 bits counter.
R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn		Generates software trigger.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channeln
r_{Config_TAU*m*_*n*}_interrupt		Executes processing in response to end of timer channel*n* count end interrupt.

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channeln module in delay counter mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Lower8bits_Start

This API function starts the TAU*m* channeln lower 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Lower8bits_Start(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Lower8bits_Stop

This API function stops the TAU*m* channeln lower 8 bits counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Lower8bits_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number 1 or 3.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn

Generates software trigger.

[Syntax]

```
void    R_{Config_TAUm_n}_Set_SoftwareTriggerOn(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channeln.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_interrupt

This API function executes processing in response to end of timer channelmn count end interrupt.

Remark This API function is called as the interrupt handler for count end interrupt, which occur when the current counter value (TCRmn) reaches 0000H.

[Syntax]

```
void    r_{Config_TAUm_n}_interrupt(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using TAU channel 0 counting as delay counter mode and channel 1 counting as 8-bit delay counter for a user-defined period:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;
extern uint8_t ch1_run_count;

int main(void);

int main(void)
{
    ch0_run_count = 0;
    R_Config_TAU0_0_Start();
    R_Config_TAU0_0_Set_SoftwareTriggerOn();
    while (ch0_run_count < 20);
    R_Config_TAU0_0_Stop();

    Ch1_run_count = 0;
    R_Config_TAU0_1_Lower8bits_Start();
    R_Config_TAU0_1_Set_SoftwareTriggerOn();
    while (ch0_run_count < 20);
    R_Config_TAU0_1_Lower8bits_Stop();

    return 0;
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_interrupt. Do not edit comment generated here */
    ch1_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.15 One-Shot Pulse Output (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for one-shot pulse output use.

Table 4-16 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in one-shot pulse output mode.
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn		Generates software trigger.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channel*n*.
r_{Config_TAU*m*_*n*}_channel*n*_interrupt		Executes processing in response to timer channel*n* count end interrupt.
r_{Config_TAU*m*_*n*}_channel*p*_interrupt		Executes processing in response to timer channel*p* count end interrupt.

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channeln module in one-shot pulse output mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn

This API function generates software trigger.

[Syntax]

```
void    R_{Config_TAUm_n}_Set_SoftwareTriggerOn(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channeln.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_channel*n*_interrupt

This API function executes processing in response to timer channelmn count end interrupt.

Remark This API function is called as the interrupt handler for count end interrupt, which occur when the current counter value (TCRmn) reaches 0000H.

[Syntax]

```
void    r_{Config_TAUm_n}_channeln_interrupt(void);
```

Remark *m* is the unit number, *n* is the master channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_channel*p*_interrupt

This API function executes processing in response to timer channelmp count end/capture interrupt.

Remark1. In one-shot pulse output function, this API function is called as the interrupt handler for count end interrupt, which occur when the current counter value (TCRmp) reaches 0000H.

Remark2. In two-channel input with one-shot pulse output function, this API function is called as the interrupt handler for capture interrupt, which occur when the valid capture edge is detected, and the current counter value (TCRmp) is transferred to timer data register mp (TDRmp).

[Syntax]

```
void    r_{Config_TAUm_n}_channelp_interrupt(void);
```

Remark1. *m* is the unit number, *n* is the master channel number, *p* is slave channel number.

Remark2. $n < p \leq 7$.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for TAU channel 0 outputting one-shot pulse by software trigger:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t tau_oneshot_count

int main(void);

int main(void)
{
	tau_oneshot_count = 0;
	R_Config_TAU0_0_Start();
	R_Config_TAU0_0_Set_SoftwareTriggerOn();
	while (tau_ oneshot_count < 10);
	R_Config_TAU0_0_Stop();

	return 0;
}
```

Config_TAU0_0_user.c

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

static void __near r_Config_TAU0_01_channel1_interrupt (void)
{
	/* Start user code for r_Config_TAU0_01_channel1_interrupt. Do not edit comment generated here
*/
	tau_oneshot_count ++;
	/* End user code. Do not edit comment generated here */
}
```

4.2.16 PWM Output (Timer Array Unit)

Below is a list of API functions output by the Smart Configurator for PWM output use.

Table 4-17 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_TAU*m*_*n*}_Create	Timer Array Unit	Executes initialization processing that is required before controlling the TAU*m* channel*n* module in PWM mode.
R_{Config_TAU*m*_*n*}_Start		Starts the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Stop		Stops the TAU*m* channel*n* counter.
R_{Config_TAU*m*_*n*}_Create_UserInit		Executes user-specific initialization processing for the TAU*m* channel*n*.
r_{Config_TAU*m*_*n*}_channel*n*_interrupt		Executes processing in response to timer channel*n* count end interrupt.
r_{Config_TAU*m*_*n*}_channel*p*_interrupt		Executes processing in response to timer channel*p* count end interrupt

R_{Config_TAU*m*_*n*}_Create

This API function executes initialization processing that is required before controlling the TAU*m* channeln module in PWM mode.

Remark This API function is called from R_TAU*m*_Create.

[Syntax]

```
void    R_{Config_TAUm_n}_Create(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Start

This API function starts the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Start(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Stop

This API function stops the TAU*m* channeln counter.

[Syntax]

```
void    R_{Config_TAUm_n}_Stop(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_TAU*m*_*n*}_Create_UserInit

This API function executes user-specific initialization processing for the TAU*m* channeln.

Remark This API functions is called from R_{Config_TAU*m*_*n*}_Create as a callback routine.

[Syntax]

```
void R_{Config_TAUm_n}_Create_UserInit(void);
```

Remark *m* is the unit number, *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_channel*n*_interrupt

This API function executes processing in response to timer channelmn count end interrupt.

Remark This API function is called as the interrupt handler for count end interrupt, which occur when the current counter value (TCRmn) reaches 0000H.

[Syntax]

```
void    r_{Config_TAUm_n}_channeln_interrupt(void);
```

Remark *m* is the unit number, *n* is the master channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_TAU*m*_*n*}_channel*p*_interrupt

This API function executes processing in response to timer channelmp count end interrupt.

Remark This API function is called as the interrupt handler for count end interrupt, which occur when the current counter value (TCRmp) reaches 0000H.

[Syntax]

```
void    r_{Config_TAUm_n}_channelp_interrupt(void);
```

Remark1. *m* is the unit number, *n* is the master channel number, *p* is slave channel number.

Remark2. $n < p \leq 7$.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for starting TAU channel 0/1 to output PWM pulses:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t tau_pwm_count;

int main(void);

int main(void)
{
    tau_pwm_count = 0;
    R_Config_TAU0_0_Start();
    while (tau_interrupt_flag < 10);
    R_Config_TAU0_0_Stop();

    return 0;
}
```

Config_TAU0_0_user.c

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

static void __near r_Config_TAU0_01_channel1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_01_channel1_interrupt. Do not edit comment generated here
*/
    tau_pwm_count++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.17 Interval Timer (32-bit Interval Timer using 8-bit counter mode)

Below is a list of API functions output by the Smart Configurator for interval timer (for 32-bit interval timer when using 8bit counter mode) use.

Table 4-18 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_ITL*n*}_Create	32-bit Interval Timer	Executes initialization processing that is required before controlling the ITL*n* module in interval timer mode (8bit mode).
R_{Config_ITL*n*}_Start		Starts the ITL*n* channel.
R_{Config_ITL*n*}_Stop		Stops the ITL*n* channel.
R_{Config_ITL*n*}_Set_OperationMode		Used to stop counter and clear interrupt flag before changing 32-bit interval timer operation mode.
R_{Config_ITL*n*}_Create_UserInit		Executes user-specific initialization processing for the ITL*n* channel.
r_{Config_ITL*n*}_Callback_Shared_Interrupt		Executes processing in response to 32-bit interval timer interrupt.

R_{Config_ITL*n*}_Create

This API function executes initialization processing that is required before controlling the ITL*n* module in interval timer mode (8bit mode).

Remark This API function is called from R_ITL_Create.

[Syntax]

```
void    R_{Config_ITLn}_Create(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*}_Start

This API function starts the ITL*n* channel.

Remark The 32-bit interval timer interrupt is enabled by calling R_ITL_Start_Interrupt. For this reason, to use 32-bit interval timer interrupt, please call this API function together with R_ITL_Start_Interrupt.

[Syntax]

```
void    R_{Config_ITLn}_Start(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*}_Stop

This API function stops the ITL*n* channel.

[Syntax]

```
void    R_{Config_ITLn}_Stop(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*}_Set_OperationMode

This API function is used to stop counter and clear interrupt flag before changing 32-bit interval timer operation mode.

[Syntax]

```
void    R_{Config_ITLn}_Set_OperationMode(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*}_Create_UserInit

This API function executes user-specific initialization processing for the ITL*n* channel.

Remark This API functions is called from R_{Config_ITL*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_ITLn}_Create_UserInit(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_ITL*n*}_Callback_Shared_interrupt

This API function executes processing in response to 32-bit interval timer interrupt.

Remark 1. This API function is called as a callback routine from r_itl_interrupt, which is the interrupt handler for 32-bit interval timer interrupts.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
void r_{Config_ITLn}_Callback_Shared_interrupt(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using 8-bit counting for a user-defined counter value and output a wave form P00 pin:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t interrupt_flag;

int main(void);

int main(void)
{
	interrupt_flag = 0;
	R_Config_ITL001 _Start();
	R_ITL_Start_Interrupt();
	while (interrupt_flag < 20);
	R_Config_ITL001 _Stop();

	return 0;
}
```

Config_ITL001 _user.c

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

void R_Config_ITL001_Callback_Shared_Interrupt(void)
{
	/* Start user code for R_Config_ITL000_Callback_Shared_Interrupt. Do not edit comment generated
here */
	char Pmn[] = "P00";
	interrupt_flag ++;

	if(interrupt_flag % 2 == 0)
	{
		R_Config_PORT_Set_Output_Level(Pmn, LOW);
	}
	else
	{
		R_Config_PORT_Set_Output_Level(Pmn, HIGH);
	}
	/* End user code. Do not edit comment generated here */
}
```

4.2.18 Interval Timer (32-bit Interval Timer using 16-bit counter mode)

Below is a list of API functions output by the Smart Configurator for interval timer (for 32-bit interval timer when using 16bit counter mode) use.

Table 4-19 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_ITL*n*_ITL*m*}_Create	32-bit Interval Timer	Executes initialization processing that is required before controlling the ITL*n*_ITL*m* module in interval timer mode (16bit mode).
R_{Config_ITL*n*_ITL*m*}_Start		Starts the ITL*n*_ITL*m* channel.
R_{Config_ITL*n*_ITL*m*}_Stop		Stops the ITL*n*_ITL*m* channel.
R_{Config_ITL*n*_ITL*m*}_Set_SoftwareTriggerOn		Generates software trigger.
R_{Config_ITLn_ITLm}_Set_OperationMode		Used to stop counter and clear interrupt flag before changing 32-bit interval timer operation mode.
R_{Config_ITL*n*_ITL*m*}_Get_CaptureValue		Gets capture value.
R_{Config_ITL*n*_ITL*m*}_Create_UserInit		Executes user-specific initialization processing for the ITL*n*_ITL*m* channel.
r_{Config_ITL*n*_ITL*m*}_Callback_Shared_Interrupt		Executes processing in response to 32-bit interval timer interrupt.

R_{Config_ITL*n*_ITL*m*}_Create

This API function executes initialization processing that is required before controlling the ITL*n*_ITL*m* module in interval timer mode (16bit mode).

Remark This API function is called from R_ITL_Create.

[Syntax]

```
void    R_{Config_ITLn_ITLm}_Create(void);
```

Remark When *n* is 000, *m* is 001; When *n* is 012, *m* is 013.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*_ITL*m*}_Start

This API function starts the ITL*n*_ITL*m* channel.

Remark The 32-bit interval timer interrupt is enabled by calling R_ITL_Start_Interrupt. For this reason, to use 32-bit interval timer interrupt, please call this API function together with R_ITL_Start_Interrupt.

[Syntax]

```
void    R_{Config_ITLn_ITLm}_Start(void);
```

Remark When *n* is 000, *m* is 001; When *n* is 012, *m* is 013.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*_ITL*m*}_Stop

This API function stops the ITL*n*_ITL*m* channel.

[Syntax]

```
void    R_{Config_ITLn_ITLm}_Stop(void);
```

Remark When *n* is 000, *m* is 001; When *n* is 012, *m* is 013.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*_ITL*m*}_Set_SoftwareTriggerOn

This API function generates software trigger.

[Syntax]

```
void    R_{Config_ITLn_ITLm}_Set_SoftwareTriggerOn(void);
```

Remark When *n* is 000, *m* is 001; When *n* is 012, *m* is 013.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*_ITL*m*}_Set_OperationMode

This API function is used to stop counter and clear interrupt flag before changing 32-bit interval timer operation mode.

[Syntax]

```
void    R_{Config_ITLn_ITLm}_Set_OperationMode(void);
```

Remark When *n* is 000, *m* is 001; When *n* is 012, *m* is 013.

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL*n*_ITL*m*}_Get_CaptureValue

This API function gets capture value.

[Syntax]

```
void    R_{Config_ITLn_ITLm}_Get_CaptureValue(uint16_t * const value);
```

Remark When *n* is 000, *m* is 001; When *n* is 012, *m* is 013.

[Argument(s)]

I/O	Argument(s)	Description
O	uint16_t * const value;	the address where to write the capture value

[Return value]

None.

R_{Config_ITL*n*_ITL*m*}_Create_UserInit

This API function executes user-specific initialization processing for the ITL*n*_ITL*m* channel.

Remark This API functions is called from R_{Config_ITL*n*_ITL*m*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_ITLn_ITLm}_Create_UserInit(void);
```

Remark When *n* is 000, *m* is 001; When *n* is 012, *m* is 013.

[Argument(s)]

None.

[Return value]

None.

r_{Config_ITL*n*_ITL*m*}_Callback_Shared_interrupt

This API function executes processing in response to 32-bit interval timer interrupt.

Remark 1. This API function is called as a callback routine from r_itl_interrupt, which is the interrupt handler for 32-bit interval timer interrupts.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
void    r_{Config_ITLn_ITLm}_Callback_Shared_interrupt(void);
```

Remark When *n* is 000, *m* is 001; When *n* is 012, *m* is 013.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for changing 32-bit interval timer operation mode to user setting (16-bit count mode change to 16-bit capture mode):
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
      R_Config_ITL000_ITL001_Set_OperationMode();
      /* Capture setting */
      R_TML32->ITLCC0 |= _80_ITL_CAPTURE_ENABLE;
      R_TML32->ITLCC0_b.CAPF0CR = 1U;
      R_TML32->ITLCC0 |= _00_ITL_CAPTURE_COUNTER_RETAIN;
      R_TML32->ITLCC0 &= _FC_ITL_CAPTURE_TRIGGER_CLEAR;
      R_TML32->ITLCC0 |= _00_ITL_CAPTURE_TRIGGER_SOFTWARE;
      R_Config_ITL000_ITL001_Start();
      R_ITL_Start_Interrupt();
      R_Config_ITL000_ITL001_Set_SoftwareTriggerOn();

      return 0;
}
```

Config_ITL000_user.c

```
volatile uint16_t value = 0U;

void r_Config_ITL000_callback_shared_interrupt(void)
{
   R_Config_ITL000_ITL001_Get_CaptureValue (&value);
}
```

4.2.19 Interval Timer (32-bit Interval Timer using 32-bit counter mode)

Below is a list of API functions output by the Smart Configurator for interval timer (for 32-bit interval timer when using 32bit counter mode) use.

Table 4-20 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_ITL000_ITL001_ITL012_ITL013}_Create	32-bit Interval Timer	Executes initialization processing that is required before controlling the ITL000_ITL001_ITL012_ITL013 module in interval timer mode (32bit mode).
R_{Config_ITL000_ITL001_ITL012_ITL013}_Start		Starts the ITL000_ITL001_ITL012_ITL013 channel.
R_{Config_ITL000_ITL001_ITL012_ITL013}_Stop		Stops the ITL000_ITL001_ITL012_ITL013 channel.
R_{Config_ITL000_ITL001_ITL012_ITL013}_Set_OperationMode		Used to stop counter and clear interrupt flag before changing 32-bit interval timer operation mode.
R_{Config_ITL000_ITL001_ITL012_ITL013}_Create_UserInit		Executes user-specific initialization processing for the ITL000_ITL001_ITL012_ITL013 channel.
r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_Interrupt		Executes processing in response to 32-bit interval timer interrupt.

R_{Config_ITL000_ITL001_ITL012_ITL013}_Create

This API function executes initialization processing that is required before controlling the ITL000_ITL001_ITL012_ITL013 module in interval timer mode (32bit mode).

Remark This API function is called from R_ITL_Create.

[Syntax]

```
void R_{Config_ITL000_ITL001_ITL012_ITL013}_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL000_ITL001_ITL012_ITL013}_Start

This API function starts the ITL000_ITL001_ITL012_ITL013 channel.

Remark The 32-bit interval timer interrupt is enabled by calling R_ITL_Start_Interrupt. For this reason, to use 32-bit interval timer interrupt, please call this API function together with R_ITL_Start_Interrupt.

[Syntax]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL000_ITL001_ITL012_ITL013}_Stop

This API function stops the ITL000_ITL001_ITL012_ITL013 channel.

[Syntax]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL000_ITL001_ITL012_ITL013}_Set_OperationMode

This API function is used to stop counter and clear interrupt flag before changing 32-bit interval timer operation mode.

[Syntax]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Set_OperationMode(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ITL000_ITL001_ITL012_ITL013}_Create_UserInit

This API function executes user-specific initialization processing for the ITL000_ITL001_ITL012_ITL013 channel.

Remark This API functions is called from R_{Config_ITL000_ITL001_ITL012_ITL013}_Create as a callback routine.

[Syntax]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_interrupt

This API function executes processing in response to 32-bit interval timer interrupt.

Remark 1. This API function is called as a callback routine from r_itl_interrupt, which is the interrupt handler for 32-bit interval timer interrupts.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
void    r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for using 32-bit count mode for a user-defined period:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t itl_run_count;

int main(void);

int main(void)
{
    R_Config_ITL000_ITL001_ITL012_ITL013_Start();
    R_ITL_Start_Interrupt();
    while (itl_run_count < 20);
    R_Config_ITL000_ITL001_ITL012_ITL013_Stop();

    return 0;
}
```

Config_ITL000_ITL001_ITL012_ITL013_user.c

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

void R_Config_ITL000_ITL001_ITL012_ITL013_Callback_Shared_Interrupt(void)
{
    /* Start user code for R_Config_ITL000_ITL001_ITL012_ITL013_Callback_Shared_Interrupt. Do not
edit comment generated here */
    itl_run_count ++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.20 Real-Time Clock

Below is a list of API functions output by the Smart Configurator for real-time clock use.

Table 4-21 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_RTC}_Create	Real-Time Clock	Executes initialization processing that is required before controlling the real-time clock module.
R_{Config_RTC}_Start		Enables the real-time clock counter.
R_{Config_RTC}_Stop		Disables the real-time clock counter.
R_{Config_RTC}_Set_HourSystem		Chooses 12-hour system or 24-hour system.
R_{Config_RTC}_Set_CounterValue		Changes the real-time clock counter value.
R_{Config_RTC}_Get_CounterValue		Reads the results of real-time clock and store them in the variables.
R_{Config_RTC}_Set_ConstPeriodInterruptOn		Enables constant-period interrupt.
R_{Config_RTC}_Set_ConstPeriodInterruptOff		Disables constant-period interrupt.
R_{Config_RTC}_Set_AlarmOn		Starts the alarm operation.
R_{Config_RTC}_Set_AlarmOff		Stops the alarm operation.
R_{Config_RTC}_Set_AlarmValue		Sets alarm value.
R_{Config_RTC}_Get_AlarmValue		Gets alarm value.
R_{Config_RTC}_Set_RTC1HZOn		Enables RTC1HZ output.
R_{Config_RTC}_Set_RTC1HZOff		Disables RTC1HZ output.
R_{Config_RTC}_Create_UserInit		Executes user-specific initialization processing for the real-time clock.
r_{Config_RTC}_interrupt		Executes processing in response to alarm, fixed-period interrupt.
r_{Config_RTC}_callback_constperiod		Executes processing in response to fixed-period interrupt.
r_{Config_RTC}_callback_alarm		Executes processing in response to alarm interrupt.

R_{Config_RTC}_Create

This API function executes initialization processing that is required before controlling the real-time clock module.

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 enables the real-time clock counter.

[Syntax]

```
void    R_{Config_RTC}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_RTC}_Stop

This API function disables the real-time clock counter.

[Syntax]

```
void    R_{Config_RTC}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_RTC}_Set_HourSystem

Chooses 12-hour system or 24-hour system.

[Syntax]

```
MD_STATUS      R_{Config_RTC}_Set_HourSystem(e_rtc_hour_system_t hour_system);
```

[Argument(s)]

I/O	Argument(s)	Description
I	e_rtc_hour_system_t hour_system;	Clock type HOUR12: 12-hour clock HOUR24: 24-hour clock

Remark Below is shown the structure e_rtc_hour_system_t (hour system).

```
typedef enum
{
    HOUR12,
    HOUR24
} e_rtc_hour_system_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_BUSY1	Busy 1.
MD_BUSY2	Busy 2.
MD_ARGERROR	Error argument input error.

R_{Config_RTC}_Set_CounterValue

Changes the real-time clock counter value.

[Syntax]

```
MD_STATUS      R_{Config_RTC}_Set_CounterValue(st_rtc_counter_value_t counter_write_val);
```

[Argument(s)]

I/O	Argument(s)	Description
I	st_rtc_counter_value_t counter_write_val;	The expected real-time clock value (BCD code)

Remark Below is shown the structure st_rtc_counter_value_t (counter conditions).

```
typedef struct
{
    uint8_t sec;
    uint8_t min;
    uint8_t hour;
    uint8_t day;
    uint8_t week;
    uint8_t month;
    uint8_t year;
} st_rtc_counter_value_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_BUSY1	Busy 1.
MD_BUSY2	Busy 2.

R_{Config_RTC}_Get_CounterValue

This API function reads the results of real-time clock and store them in the variables.

[Syntax]

```
MD_STATUS R_{Config_RTC}_Get_CounterValue(st_rtc_counter_value_t * const
counter_read_val);
```

[Argument(s)]

I/O	Argument(s)	Description
I	st_rtc_counter_value_t * const counter_read_val;	The current real-time clock value (BCD code)

Remark For structure st_rtc_counter_value_t, see R_{Config_RTC}_Set_CounterValue.

[Return value]

Macro	Description
MD_OK	Normal end
MD_BUSY1	Busy 1.
MD_BUSY2	Busy 2.

R_{Config_RTC}_Set_ConstPeriodInterruptOn

Enables constant-period interrupt.

[Syntax]

```
MD_STATUS      R_{Config_RTC}_Set_ConstPeriodInterruptOn(e_rtc_int_period_t period);
```

[Argument(s)]

I/O	Argument(s)	Description
I	e_rtc_int_period_t period;	The fixed-cycle period enumeration

Remark Below is shown the structure e_rtc_int_period_t period (period conditions).

```
typedef enum
{
    HALFSEC = 1U,
    ONESEC,
    ONEMIN,
    ONEHOUR,
    ONEDAY,
    ONEMONTH
} e_rtc_int_period_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_RTC}_Set_ConstPeriodInterruptOff

Disables constant-period interrupt.

[Syntax]

```
void    R_{Config_RTC}_Set_ConstPeriodInterruptOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_RTC}_Set_AlarmOn

This API function starts the alarm operation.

[Syntax]

```
void    R_{Config_RTC}_Set_AlarmOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_RTC}_Set_AlarmOff

This API function stops the alarm operation.

[Syntax]

```
void    R_{Config_RTC}_Set_AlarmOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_RTC}_Set_AlarmValue

This API function sets alarm value.

[Syntax]

```
void    R_{Config_RTC}_Set_AlarmValue(st_rtc_alarm_value_t alarm_val);
```

[Argument(s)]

I/O	Argument(s)	Description
I	st_rtc_alarm_value_t alarm_val;	The expected alarm value (BCD code)

Remark Below is shown the structure st_rtc_alarm_value_t alarm_val (alarm conditions).

```
typedef struct
{
    uint8_t alarmwm;
    uint8_t alarmwh;
    uint8_t alarmww;
} st_rtc_alarm_value_t;
```

[Return value]

None.

R_{Config_RTC}_Get_AlarmValue

Gets alarm value.

[Syntax]

```
void    R_{Config_RTC}_Get_AlarmValue(st_rtc_alarm_value_t * const alarm_val);
```

[Argument(s)]

I/O	Argument(s)	Description
O	st_rtc_alarm_value_t * const alarm_val;	The address to save alarm value (BCD code)

Remark For structure st_rtc_alarm_value_t * const alarm_val,
see R_{Config_RTC}_Set_AlarmValue.

[Return value]

None.

R_{Config_RTC}_Set_RTC1HZOn

Enables RTC1HZ output.

[Syntax]

```
void    R_{Config_RTC}_Set_RTC1HZOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_RTC}_Set_RTC1HZOff

Disables RTC1HZ output.

[Syntax]

```
void    R_{Config_RTC}_Set_RTC1HZOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_RTC}_Create_UserInit

This API function executes user-specific initialization processing for the real-time clock.

Remark This API functions 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}_interrupt

This API function executes processing in response to alarm, constant period interrupt.

[Syntax]

```
void    r_{Config_RTC}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_RTC}_callback_constperiod

This API function executes processing in response to fixed-cycle interrupt.

Remark This API function is called as the callback routine of interrupt process r_{Config_RTC}_interrupt corresponding to the fixed-cycle interrupt.

[Syntax]

```
static void     r_{Config_RTC}_callback_constperiod(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_RTC}_callback_alarm

This API function executes processing in response to alarm interrupt.

Remark This API function is called as the callback routine of interrupt process r_{Config_RTC}_interrupt corresponding to the alarm interrupt.

[Syntax]

```
static void     r_{Config_RTC}_callback_alarm(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example 1 (alarm interrupt)

This is an example for 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):
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    R_Config_RTC_Set_AlarmOn();
    R_Config_RTC_Start();

    return 0;
}
```

Config_RTC_user.c

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

static void r_Config_RTC_callback_alarm(void)
{
    /* Start user code for r_Config_RTC_callback_alarm. Do not edit comment generated here */
    R_Config_RTC_Get_CounterValue ((st_rtc_counter_value_t *)&counter_val);

    /* Change the seconds */
    counter_val.rseccnt = 0x58U;

    R_Config_RTC_Set_CounterValue (counter_val);
    /* End user code. Do not edit comment generated here */
}
```

Usage example 2 (constant-period interrupt)

This is an example for using constant-period interrupts to implement generating an alarm interrupt every 1 hour: (Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
st_rtc_counter_value_t    currTime;
st_rtc_alarm_value_t alarm;

int main(void);

int main(void)
{
      R_Config_RTC_Set_ConstPeriodInterruptOn(ONEHOUR);
      R_Config_RTC_Start();

      return 0;
}
```

Config_RTC_user.c

```
/* Start user code for global. Do not edit comment generated here */
st_rtc_counter_value_t    currTime;
st_rtc_alarm_value_t alarm;
/* End user code. Do not edit comment generated here */

static void r_Config_RTC_callback_constperiod (void)
{
      /* Start user code for r_Config_RTC_callback_constperiod. Do not edit comment generated here */
      R_Config_RTC_Get_CounterValue(&currTime);
      R_Config_RTC_Get_AlarmValue(&alarm);
      alarm.alarmww = currTime.week;
      alarm.alarmwh = currTime.hour;
      alarm.alarmwm = currTime.min + 5;
      R_Config_RTC_Set_AlarmValue(alarm);
      R_Config_RTC_Set_AlarmOn();
      /* End user code. Do not edit comment generated here */
}
```

4.2.21 Watchdog Timer

Below is a list of API functions output by the Smart Configurator for watchdog timer use.

Table 4-22 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_WDT}_Create	Watchdog Timer	Executes initialization processing that is required before controlling the watchdog timer module.
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.
r_{Config_WDT}_interrupt		Executes processing in response to maskable WDT interrupt.

R_{Config_WDT}_Create

This API function executes initialization processing that is required before controlling the watchdog timer module.

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.

Remark This API functions 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}_interrupt

This API function executes processing in response to maskable WDT interrupt.

[Syntax]

```
void    r_{Config_WDT}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for refreshing the counter value on every loop of the main function and issuing a software reset in response to an underflow of the counter:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    while (1U)
    {
        /* Restarts WDT module */
        R_Config_WDT_Restart();
    }

    return 0;
}
```

4.2.23 Independent Watchdog Timer

Below is a list of API functions output by the Smart Configurator for independent watchdog timer use.

Table 4-23 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_IWDT}_Create	Watchdog Timer	Executes initialization processing that is required before controlling the watchdog timer module.
R_{Config_IWDT}_Restart		Clears the counter in the watchdog timer, and then restarts counting by the counter.
R_{Config_IWDT}_Create_UserInit		Executes user-specific initialization processing for the watchdog timer.
r_{Config_IWDT}_interrupt		Executes processing in response to maskable IWDT interrupt.

R_{Config_IWDT}_Create

This API function executes initialization processing that is required before controlling the watchdog timer module.

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

[Syntax]

```
void    R_{Config_IWDT}_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_IWDT}_Restart

This API function clears the counter in the watchdog timer, and then restarts counting by the counter.

[Syntax]

```
void    R_{Config_IWDT}_Restart(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_IWDT}_Create_UserInit

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

Remark This API functions is called from R_{Config_IWDT}_Create as a callback routine.

[Syntax]

```
void    R_{Config_IWDT}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_IWDT}_interrupt

This API function executes processing in response to maskable IWDT interrupt.

[Syntax]

```
void    r_{Config_IWDT}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for refreshing the counter value on every loop of the main function and issuing a software reset in response to an underflow of the counter:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    while (1U)
    {
        /* Restarts WDT module */
        R_Config_IWDT_Restart();
    }

    return 0;
}
```

4.2.24 Simplified SPI Communication

Below is a list of API functions output by the Smart Configurator for Simplified SPI communication use.

Table 4-24 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_SPI*p*}_Create	Serial Array Unit	Executes initialization processing that is required before controlling the SPI*p* module.
R_{Config_SPI*p*}_Start		Starts the SPI*p* module operation.
R_{Config_SPI*p*}_Stop		Stops the SPI*p* module operation.
R_{Config_SPI*p*}_Send		Sends SPI*p* data.
R_{Config_SPI*p*}_Receive		Receives SPI*p* data.
R_{Config_SPI*p*}_Send_Receive		Sends and receives SPI*p* data.
R_{Config_SPI*p*}_Create_UserInit		Executes user-specific initialization processing for the SPI*p*.
r_{Config_SPI*p*}_interrupt		Executes processing in response to transfer end interrupt/buffer empty interrupt.
r_{Config_SPI*p*}_callback_sendend		Eexecutes processing in response to transmit end interrupt.
r_{Config_SPI*p*}_callback_receiveend		Executes processing in response to receive end interrupt.
r_{Config_SPI*p*}_callback_error		Executes processing in response to occur transfer error.

R_{Config_SPI*p*}_Create

This API function executes initialization processing that is required before controlling the SPI*p* module.

Remark1. This API function is called from R_SAU*m*_Create.

Remark2. When *m* is 0, *p* is 00, 01, 10, 11; When *m* is 1, *p* is 20, 21.

[Syntax]

```
void    R_{Config_SPIp}_Create(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

R_{Config_SPI*p*}_Start

This API function starts the SPI*p* module operation.

[Syntax]

```
void    R_{Config_SPIp}_Start(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

R_{Config_SPI*p*}_Stop

This API function stops the SPI*p* module operation.

[Syntax]

```
void    R_{Config_SPIp}_Stop(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

R_{Config_SPI*p*}_Send

This API function sends SPI*p* data.

[Syntax]

```
MD_STATUS      R_{Config_SPIp}_Send(uint8_t * const tx_buf, uint16_t tx_num);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t * const tx_buf;	Transfer buffer pointer
I	uint16_t tx_num;	Buffer size

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_SPI*p*}_Receive

This API function receives SPIp data.

[Syntax]

```
MD_STATUS      R_{Config_SPIp}_Receive(uint8_t * const rx_buf, uint16_t rx_num);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

I/O	Argument(s)	Description
O	uint8_t * const rx_buf;	Receive buffer pointer
I	uint16_t rx_num;	Buffer size

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_SPI*p*}_Send_Receive

Sends and receives SPI*p* data.

[Syntax]

```
MD_STATUS R_{Config_SPIp}_Send_Receive(uint8_t * const tx_buf, uint16_t tx_num, uint8_t * const rx_buf);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t * const tx_buf;	Transfer buffer pointer
O	uint8_t * const rx_buf;	Receive buffer pointer
I	uint16_t tx_num;	Buffer size

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_SPI*p*}_Create_UserInit

This API function executes user-specific initialization processing for the SPI*p*.

Remark This API functions is called from R_{Config_SPI*p*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_SPIp}_Create_UserInit(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

r_{Config_SPI*p*}_interrupt

This API function executes processing in response to transfer end interrupt/buffer empty interrupt.

[Syntax]

```
void    r_{Config_SPIp}_interrupt(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

r_{Config_SPI*p*}_callback_sendend

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

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_SPI*p*}_interrupt corresponding to the SPI*p* interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_SPIp}_callback_sendend(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

r_{Config_SPI*p*}_callback_receiveend

This API function executes processing in response to receive end interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_SPI*p*}_interrupt corresponding to the SPI*p* interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_SPIp}_callback_receiveend(void);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

r_{Config_SPI*p*}_callback_error

This API function executes processing in response to occur transfer error.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_SPI*p*}_interrupt corresponding to the SPI*p* interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_SPIp}_callback_error(uint8_t err_type);
```

Remark *p* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t err_type;	Error type value Bit0: Overrun error

[Return value]

None.

Usage example

This is an example for SPI00 send data and SPI11 receive these data in continuous mode:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf[6] = {0xA5,0x3F,0xC0,0x5C,0xB6,0x37};
uint8_t rx_buf[6] = {0x00,0x00,0x00,0x00,0x00,0x00};
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

int main(void);

int main(void)
{
    R_Config_SPI11_Start();
    R_Config_SPI00_Start();
    R_Config_SPI11_Receive(rx_buf, sizeof(rx_buf));
    R_Config_SPI00_Send(tx_buf, sizeof(tx_buf));
    while(transmitend_flag != 1U);
    transmitend_flag = 0U;
    while(receiveend_flag != 1U);
    receiveend_flag = 0U;
    R_Config_SPI00_Stop();
    R_Config_SPI11_Stop();

    return 0;
}
```

Config_SPI00_user.c

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

static void r_Config_SPI00_callback_sendend(void)
{
    /* Start user code for r_Config_SPI00_callback_sendend. Do not edit comment generated here */
    transmitend_flag = 1U;
    /* End user code. Do not edit comment generated here */
}
```

Config_SPI11_user.c

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

static void r_Config_SPI11_callback_receiveend (void)
{
    /* Start user code for r_Config_SPI11_callback_receiveend. Do not edit comment generated here */
    receiveend_flag = 1U;
    /* End user code. Do not edit comment generated here */
}
```

4.2.25 UART Communication (Serial array unit)

Below is a list of API functions output by the Smart Configurator for UART Communication use.

Table 4-25 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_UART*q*}_Create	Serial Array Unit	Executes initialization processing that is required before controlling the UART*q* module.
R_{Config_UART*q*}_Start		Starts UART*q* module operation.
R_{Config_UART*q*}_Stop		Stops UART*q* module operation.
R_{Config_UART*q*}_Send		Sends UART*q* data.
R_{Config_UART*q*}_Receive		Receives UART*q* data.
R_{Config_UART*q*}_Loopback_Enable		Enables the UART*q* loopback function.
R_{Config_UART*q*}_Loopback_Disable		Disables the UART*q* loopback function.
R_{Config_UART*q*}_Create_UserInit		Executes user-specific initialization processing for the UART*q*.
r_{Config_UART*q*}_interrupt_send		Executes processing in response to UART*q* transmit end interrupt (in single-transfer mode) or buffer empty interrupt (in continuous transfer mode).
r_{Config_UART*q*}_interrupt_receive		Executes processing in response to UART*q* receive end interrupt.
r_{Config_UART*q*}_interrupt_error		Executes processing in response to UART*q* error interrupt.
r_{Config_UART*q*}_callback_sendend		Executes processing in response to transmit end interrupt.
r_{Config_UART*q*}_callback_receiveend		Executes processing in response to receive end interrupt.
r_{Config_UART*q*}_callback_error		Executes processing in response to receive error interrupt.
r_{Config_UART*q*}_callback_softwareoverrun		Executes processing in response to receive an overflow data.

R_{Config_UART*q*}_Create

This API function executes initialization processing that is required before controlling the UART*q* module.

Remark1. This API function is called from R_SAU*m*_Create.

Remark2. When *m* is 0, *q* is 0, 1; When *m* is 1, *q* is 2, 3.

[Syntax]

```
void    R_{Config_UARTq}_Create(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UART*q*}_Start

This API function starts UART*q* module operation.

[Syntax]

```
void    R_{Config_UART q}_Start(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UART*q*}_Stop

This API function stops UART*q* module operation.

[Syntax]

```
void    R_{Config_UART q}_Stop(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UART*q*}_Send

This API function sends UART*q* data.

[Syntax]

```
MD_STATUS      R_{Config_UARTq}_Send(uint8_t * const tx_buf, uint16_t tx_num);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t * const tx_buf;	Transfer buffer pointer
I	uint16_t tx_num;	Buffer size

[Return value]

Macro	Description
MD_OK	Normal end (send the first data)
MD_ARGERROR	Error argument input error.

R_{Config_UART*q*}_Receive

This API function receives UART*q* data.

[Syntax]

```
MD_STATUS      R_{Config_UARTq}_Receive(uint8_t * const rx_buf, uint16_t rx_num);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

I/O	Argument(s)	Description
O	uint8_t * const rx_buf;	Receive buffer pointer
I	uint16_t rx_num;	Buffer size

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_UART*q*}_Loopback_Enable

This API function enables the UART*q* loopback function.

[Syntax]

```
void    R_{Config_UARTq}_Loopback_Enable(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UART*q*}_Loopback_Disable

This API function disables the UART*q* loopback function.

[Syntax]

```
void    R_{Config_UARTq}_Loopback_Disable(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UART*q*}_Create_UserInit

This API function executes user-specific initialization processing for the UART*q*.

Remark This API functions is called from R_{Config_UART*q*}_Create as a callback routine.

[Syntax]

```
void R_{Config_UARTq}_Create_UserInit(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UART*q*}_interrupt_send

This API function executes processing in response to UART*q* transmit end interrupt (in single-transfer mode) or buffer empty interrupt (in continuous transfer mode).

[Syntax]

```
void    r_{Config_UARTq}_interrupt_send(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UART*q*}_interrupt_receive

This API function executes processing in response to UART*q* receive end interrupt.

[Syntax]

```
void    r_{Config_UART q}_interrupt_receive(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UART*q*}_interrupt_error

This API function executes processing in response to UART*q* error interrupt.

[Syntax]

```
void    r_{Config_UARTq}_interrupt_error(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UART*q*}_callback_sendend

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

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_UART*q*}_interrupt_send corresponding to the UART*q* transmit end interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_UARTq}_callback_sendend(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UART*q*}_callback_receiveend

This API function executes processing in response to receive end interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_UART*q*}_interrupt_receive corresponding to the UART*q* receive end interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_UARTq}_callback_receiveend(void);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UART*q*}_callback_error

This API function executes processing in response to receive error interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_UART*q*}_interrupt_error corresponding to the UART*q* receive error interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_UARTq}_callback_error(uint8_t err_type);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t err_type;	Error type info Bit0: Overrun error Bit1: Parity error Bit2: Framing error Bit3 to Bit7: 0

[Return value]

None.

r_{Config_UART*q*}_callback_softwareoverrun

This API function executes processing in response to receive an overflow data.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_UART*q*}_interrupt_receive corresponding to the UARTq receive end interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_UARTq}_callback_softwareoverrun(uint16_t rx_data);
```

Remark *q* is 0, 1, 2.

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t rx_data;	Receive data

[Return value]

None.

Usage example

This is an example for UART0 send data and UART1 receive these data:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf[6] = {0xA5,0x3F,0xC0,0x5C,0xB6,0x37};
uint8_t rx_buf[6] = {0x00,0x00,0x00,0x00,0x00,0x00};
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

int main(void);

int main(void)
{

      R_Config_UART0_Start();
      R_Config_UART1_Start();
      R_Config_UART1_Receive(rx_buf, 6);
      R_Config_UART0_Send(tx_buf, 6);
      while(transmitend_flag != 1U && receiveend_flag != 1U);
      transmitend_flag = 0U;
      receiveend_flag = 0U;
      R_Config_UART0_Stop();
      R_Config_UART1_Stop();
}
```

Config_UART0_user.c

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

static void r_Config_UART0_callback_sendend(void)
{
      /* Start user code for r_Config_UART0_callback_sendend. Do not edit comment generated here */
      transmitend_flag = 1U;
      /* End user code. Do not edit comment generated here */
}
```

Config_UART1_user.c

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

static void r_Config_UART1_callback_receiveend (void)
{
      /* Start user code for r_Config_UART1_callback_receiveend. Do not edit comment generated here
*/
      receiveend_flag = 1U;
      /* End user code. Do not edit comment generated here */
}
```

4.2.26 Simplified I2C Communication (Master mode) (Serial Array Unit)

Below is a list of API functions output by the Smart Configurator for simplified I2C communication (master mode) (serial array unit) use.

Table 4-26 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_IIC*r*}_Create	Serial Array Unit	Executes initialization processing that is required before controlling the IIC*r* master module.
R_{Config_IIC*r*}_StartCondition		Issues a start condition.
R_{Config_IIC*r*}_StopCondition		Issues a stop condition.
R_{Config_IIC*r*}_Stop		Stops the IIC*r* module.
R_{Config_IIC*r*}_Master_Send		Starts transferring data for IIC*r* in master mode.
R_{Config_IIC*r*}_Master_Receive		Starts receiving data for IIC*r* in master mode.
R_{Config_IIC*r*}_Create_UserInit		Executes user-specific initialization processing for the IIC*r*.
r_{Config_IIC*r*}_interrupt		Executes processing in response to IIC*r* transfer end interrupt.
r_{Config_IIC*r*}_callback_master_sendend		Executes processing in response to master transmit end interrupt.
r_{Config_IIC*r*}_callback_master_receiveend		Executes processing in response to master receive completed interrupt.
r_{Config_IIC*r*}_callback_master_error		Executes processing in response to the detection of an overrun or NACK error.

R_{Config_IIC*r*}_Create

This API function executes initialization processing that is required before controlling the IIC*r* master module.

Remark1. This API function is called from R_SAU*m*_Create.

Remark2. When *m* is 0, *r* is 00, 01, 10, 11; When *m* is 1, *r* is 20, 21.

[Syntax]

```
void    R_{Config_IICr}_Create(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IIC*r*}_StartCondition

This API function issues a start condition.

Remark This API function is used as an internal function of R_{Config_IICr}_Master_Send and R_{Config_IICr}_Master_Receive . For this reason, there is normally no need to call it from a user program.

[Syntax]

```
void    R_{Config_IIC r}_StartCondition(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IIC*r*}_StopCondition

This API function issues a stop condition.

[Syntax]

```
void    R_{Config_IICr}_StopCondition(void);
```

Remark *nm* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IIC*r*}_Stop

This API function stops the IIC*r* module.

[Syntax]

```
void    R_{Config_IICr}_Stop(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IIC*r*}_Master_Send

This API function starts transferring data for IIC*r* in master mode.

Remark 1. This API function repeats the byte-level simple IIC master transmission from the buffer specified in argument *tx_buf* the number of times specified in argument *tx_num*.

Remark 2. Before calling this API, please check that communiacation is stopped/suspended and SDA/SCL are High level.

[Syntax]

```
void    R_{Config_IIC r}_Master_Send(uint8_t adr, uint8_t * const tx_buf, uint16_t tx_num);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t adr;	Set address for select slave
I	uint8_t * const tx_buf;	Pointer to a buffer storing the transmission data
I	uint16_t tx_num;	Total amount of data to send

[Return value]

None.

R_{Config_IIC*r*}_Master_Receive

This API function starts receiving data for IIC*r* in master mode.

Remark 1. This API function performs byte-level simple IIC master reception the number of times specified by the argument *rx_num*, and stores the data in the buffer specified by the argument *rx_buf*.

Remark 2. Before calling this API, please check that communiacation is stopped/suspended and SDA/SCL are High level.

[Syntax]

```
void    R_{Config_IIC r}_Master_Receive(uint8_t adr, uint8_t * const rx_buf, uint16_t rx_num);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t adr;	Set address for select slave
O	uint8_t * const rx_buf;	Pointer to a buffer to store the received data
O	uint16_t rx_num;	Total amount of data to receive

[Return value]

None.

R_{Config_IIC*r*}_Create_UserInit

This API function executes user-specific initialization processing for the IIC*r*.

Remark This API functions is called from R_{Config_IIC*r*}_Create as a callback routine.

[Syntax]

```
void R_{Config_IICr}_Create_UserInit(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IIC*r*}_interrupt

This API function executes processing in response to IIC*r* transfer end interrupt.

[Syntax]

```
void    r_{Config_IIC r}_interrupt(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IIC*r*}_callback_master_sendend

This API function executes processing in response to master transmit end interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_IICr}_interrupt corresponding to the IIC*r* master transmit end interrupt.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void    r_{Config_IIC r}_callback_master_sendend(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IIC*r*}_callback_master_receiveend

This API function executes processing in response to master receive completed interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_IICr}_interrupt corresponding to the IIC*r* master receive completed interrupt.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void     r_{Config_IIC r}_callback_master_receiveend(void);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IIC*r*}_callback_master_error

This API function executes processing in response to the detection of an overrun or NACK error.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_IICr}_interrupt corresponding to the IIC*r* transmit error.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void     r_{Config_IIC r}_callback_error(MD_STATUS flag);
```

Remark *r* is 00, 01, 10, 11, 20, 21.

[Argument(s)]

I/O	Argument(s)	Description
I	MD_STATUS flag;	Error type: MD_NACK: Detection of NACK error MD_OVERRUN: Detection of overrun error

[Return value]

None.

Usage example

This is an example for IIC0 master communication with IICA0 slave (including IIC0 master send, and master receive mode):

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf[6] = {0xA5,0x3F,0xC0,0x5C,0xB6,0x37};
uint8_t rx_buf1[6] = {0x00,0x00,0x00,0x00,0x00,0x00};
uint8_t rx_buf2[6] = {0x00,0x00,0x00,0x00,0x00,0x00};
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

int main(void);

int main(void)
{
      R_Config_IIC00_StartCondition();

      R_Config_IIC00_Master_Receive(0x24,rx_buf1,sizeof(rx1_buf));
      R_Config_IICA1_Slave_Send(tx_buf,sizeof(tx_buf));

      while(receiveend_flag != 1);
      transmitend = 0;
      receiveend = 0;

      R_Config_IIC00_StopCondition();
      R_Config_IIC00_StartCondition();

      R_Config_IICA1_Slave_Receive(rx_buf2, sizeof(rx_buf2));
      R_Config_IIC00_Master_Send(0x24, tx_buf,sizeof(tx_buf));

      while(receiveend_flag != 1);
      transmitend_flag = 0;
      receiveend_flag = 0;

      R_Config_IIC00_StopCondition();
      R_Config_IIC00_Stop();
      R_Config_IICA0_Stop();

      return 0;
}
```

Config_IIC00_user.c

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

static void r_Config_IIC00_callback_master_sendend (void)
{
    /* Start user code for r_Config_IIC00_callback_master_sendend. Do not edit comment generated here */
    transmitend_flag = 1U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_IIC00_callback_master_receiveend (void)
{
    /* Start user code for r_Config_IIC00_callback_master_receiveend. Do not edit comment generated here */
    receiveend_flag = 1U;
    /* End user code. Do not edit comment generated here */
}
```

Config_IICA0_user.c

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

static void r_Config_IICA0_callback_slave_sendend(void)
{
    /* Start user code for r_Config_IICA0_callback_slave_sendend. Do not edit comment generated here */
    transmitend_flag = 1U
    /* End user code. Do not edit comment generated here */
}

static void r_Config_IICA0_callback_slave_receiveend(void)
{
    /* Start user code for r_Config_IICA0_callback_slave_receiveend. Do not edit comment generated here */
    receiveend_flag = 1U;
    /* End user code. Do not edit comment generated here */
}
```

4.2.27 I2C Bus Interface Communication (Master mode)

Below is a list of API functions output by the Smart Configurator for I2C bus interface communication (master mode) use.

Table 4-27 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_IICA*n*}_Create	Serial Interface IICA	Executes initialization processing that is required before controlling the IICAn master module.
R_{Config_IICA*n*}_StopCondition		Issues a stop condition.
R_{Config_IICA*n*}_Stop		Stops the IICAn master operation.
R_{Config_IICA*n*}_Master_Send		Starts transferring data in master mode.
R_{Config_IICA*n*}_Master_Receive		Starts receiving data in master mode.
R_{Config_IICA*n*}_Create_UserInit		Executes user-specific initialization processing for the IICAn.
r_{Config_IICA*n*}_interrupt		Executes processing in response to end of IICAn communication interrupt.
r_{Config_IICA*n*}_master_handler		Controls IICAn data transmission / reception / error in master mode.
r_{Config_IICA*n*}_callback_master_sendend		Executes processing in response to master transmit end.
r_{Config_IICA*n*}_callback_master_receiveend		Executes processing in response to master receive completed.
r_{Config_IICA*n*}_callback_master_error		Executes processing in response to the detection of a bus busy or NACK error.

R_{Config_IICA*n*}_Create

This API function executes initialization processing that is required before controlling the IICA*n* master module.

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

[Syntax]

```
void    R_{Config_IICAn}_Create(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IICA*n*}_StopCondition

This API function issues a stop condition.

[Syntax]

```
void    R_{Config_IICAn}_StopCondition(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IICA*n*}_Stop

This API function stops the IICA*n* master operation.

[Syntax]

```
void    R_{Config_IICAn}_Stop(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IICA*n*}_Master_Send

This API function starts transferring data in master mode.

[Syntax]

```
void R_{Config_IICAn}_Master_Send(uint8_t adr, uint8_t * const tx_buf, uint16_t tx_num, uint8_t wait);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t adr;	Transfer address
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
I	uint8_t wait;	Wait for start condition

[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_IICA*n*}_Master_Receive

This API function starts receiving data in master mode.

[Syntax]

```
void R_{Config_IICAn}_Master_Receive(uint8_t adr, uint8_t * const rx_buf, uint16_t rx_num, uint8_t wait);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t adr;	Receive address
O	uint8_t * const rx_buf;	Pointer to the buffer where the received data are to be stored
O	uint16_t rx_num;	Number of bytes to be received
	uint8_t wait	Wait for start condition

[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_IICA*n*}_Create_UserInit

This API function executes user-specific initialization processing for the IICA*n*.

Remark This API functions is called from R_{Config_IICA*n*}_Create as a callback routine.

[Syntax]

```
void R_{Config_IICAn}_Create_UserInit(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_interrupt

This API function executes processing in response to end of IICA0 communication interrupt.

[Syntax]

```
void    r_{Config_IICAn}_interrupt(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_master_handler

This API function controls IICA*n* data transmission / reception / error in master mode.

Remark This API function is called as the callback routine of interrupt process r_{Config_IICA*n*}_interrupt corresponding to the IICAn interrupt.

[Syntax]

```
void    R_{Config_IICAn}_master_handler(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_callback_master_sendend

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

Remark 1. This API function is called as the callback routine of interrupt process R_{Config_IICAn}_master_handler.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void     r_{Config_IICA n}_callback_master_sendend(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_callback_master_receiveend

This API function executes processing in response to master receive completed.

Remark 1. This API function is called as the callback routine of interrupt process R_{Config_IICAn}_master_handler.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void     r_{Config_IICA n}_callback_master_receiveend(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_callback_master_error

This API function executes processing in response to the detection of a bus busy or NACK error.

Remark 1. This API function is called as the callback routine of interrupt process R_{Config_IICAn}_master_handler.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void     r_{Config_IICA n}_callback_master_error(MD_STATUS flag);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
I	MD_STATUS flag;	Status flag MD_NACK: NACK error MD_SPT: BUS busy error

[Return value]

None.

Usage example

This is an example for IICA0 master communication with IICA1 slave (including both send and receive mode):
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf[6] = {0xA5,0x3F,0xC0,0x5C,0xB6,0x37};
uint8_t rx_buf1[6] = {0x00,0x00,0x00,0x00,0x00,0x00};
uint8_t rx_buf2[6] = {0x00,0x00,0x00,0x00,0x00,0x00};
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

int main(void);

int main(void)
{
      R_Config_IICA1_Slave_Receive(rx_buf1, sizeof(tx_buf));
      R_Config_IICA0_Master_Send(0x24, tx_buf,sizeof(tx_buf),100);

      while(receiveend_flag != 1);
      transmitend_flag = 0;
      receiveend_flag = 0;

      R_Config_IICA0_Master_Receive(0x24, rx_buf2, sizeof(rx_buf2),100);
      R_Config_IICA1_Slave_Send(tx_buf, sizeof(tx_buf));

      while(receiveend_flag != 1);
      transmitend_flag = 0;
      receiveend_flag = 0;

      R_Config_IICA0_Stop();
      R_Config_IICA1_Stop();

      return 0;
}
```

Config_IICA0_user.c

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

static void r_Config_IICA0_callback_master_sendend (void)
{
    SPT0 = 1U;
    /* Start user code for r_Config_IICA0_callback_master_sendend. Do not edit comment generated here */
    transmitend_flag = 1U;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_IICA0_callback_master_receiveend (void)
{
    SPT0 = 0U;
    /* Start user code for r_Config_IICA0_callback_master_receiveend. Do not edit comment generated here */
    receiveend_flag = 1U;
    /* End user code. Do not edit comment generated here */
}
```

Config_IICA1_user.c

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

static void r_Config_IICA1_callback_slave_sendend(void)
{
    /* Start user code for r_Config_IICA1_callback_slave_sendend. Do not edit comment generated here */
    transmitend_flag = 1U
    /* End user code. Do not edit comment generated here */
}

static void r_Config_IICA1_callback_slave_receiveend(void)
{
    /* Start user code for r_Config_IICA1_callback_slave_receiveend. Do not edit comment generated here */
    receiveend_flag = 1U;
    /* End user code. Do not edit comment generated here */
}
```

4.2.28 I2C Bus Interface Communication (Slave mode)

Below is a list of API functions output by the Smart Configurator for I2C bus interface communication (slave mode) use.

Table 4-28 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_IICA*n*}_Create	Serial Interface IICA	Executes initialization processing that is required before controlling the IICA*n* slave module.
R_{Config_IICA*n*}_Stop		Stops IICA*n* slave operation.
R_{Config_IICA*n*}_Slave_Send		Starts transferring data in slave mode.
R_{Config_IICA*n*}_Slave_Receive		Starts receiving data in slave mode.
R_{Config_IICA*n*}_Set_WakeupOn		Enables operation of address match wakeup function in STOP mode.
R_{Config_IICA*n*}_Set_WakeupOff		Disables operation of address match wakeup function in STOP mode.
R_{Config_IICA*n*}_Create_UserInit		Executes user-specific initialization processing for the IICA*n*.
r_{Config_IICA*n*}_interrupt		Executes processing in response to end of IICA0 communication interrupt.
r_{Config_IICA*n*}_slave_handler		Controls IICA*n* data transmission / reception / error in slave mode.
r_{Config_IICA*n*}_callback_slave_sendend		Executes processing in response to slave transmit end.
r_{Config_IICA*n*}_callback_slave_receiveend		Executes processing in response to slave receive completed.
r_{Config_IICA*n*}_callback_slave_error		Executes processing in response to the detection of an addresses not match or NACK error.
r_{Config_IICA*n*}_callback_getstopcondition		Executes processing in response to IICAn get a slave stop condition.

R_{Config_IICA*n*}_Create

This API function executes initialization processing that is required before controlling the IICAn slave module.

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

[Syntax]

```
void    R_{Config_IICAn}_Create(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IICA*n*}_Stop

This API function stops IICA*n* slave operation.

[Syntax]

```
void    R_{Config_IICAn}_Stop(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IICA*n*}_Slave_Send

This API function starts transferring data in slave mode.

Remark For the case of the master restarts without issuing a stop condition when communication is completed, be careful to take note to call the corresponding slave function on slave device. For example, on master device, R_{Config_IICAn}_Master_Receive function is called to restart communication, while R_{Config_IICAn}_Slave_Send function is called on slave device. In other words, Master and Slave API should be called in pair, otherwise, the IICA operaion is not guaranteed.

[Syntax]

```
void R_{Config_IICAn}_Slave_Send(uint8_t * const tx_buf, uint16_t tx_num);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	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]

None.

R_{Config_IICA*n*}_Slave_Receive

This API function starts receiving data in slave mode.

Remark For the case of the master restarts without issuing a stop condition when communication is completed, be careful to take note to call the corresponding slave function on slave device. For example, on master device, R_{Config_IICAn}_Master_Send function is called to restart communication, while R_{Config_IICAn}_Slave_Receive function is called on slave device. In other words, Master and Slave API should be called in pair, otherwise, the IICA operaion is not guaranteed.

[Syntax]

```
void R_{Config_IICAn}_Slave_Receive(uint8_t * const rx_buf, uint16_t rx_num);
```

Remark *n* is the channel number.

[Argument(s)]

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

[Return value]

None.

R_{Config_IICA*n*}_Set_WakeupOn

This API function enables operation of address match wakeup function in STOP mode.

[Syntax]

```
void    R_{Config_IICAn}_Set_WakeupOn(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IICA*n*}_Set_WakeupOff

This API function disables operation of address match wakeup function in STOP mode.

[Syntax]

```
void    R_{Config_IICAn}_Set_WakeupOff(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_IICA*n*}_Create_UserInit

This API function executes user-specific initialization processing for the IICA*n*.

Remark This API functions is called from R_{Config_IICA*n*}_Create as a callback routine.

[Syntax]

```
void R_{Config_IICAn}_Create_UserInit(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_interrupt

This API function executes processing in response to end of IICA0 communication interrupt.

[Syntax]

```
void    r_{Config_IICAn}_interrupt(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_slave_handler

This API function controls IICA*n* data transmission / reception / error in slave mode.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_IICA*n*}_interrupt corresponding to the IICAn interrupt.

Remark 2. Smart Configurator use "g_iica*n*_slave_status_flag" to control user program flow. The initialization of "g_iica0_slave_status_flag" is in R_{Config_IICA*n*}_Slave_Send function and R_{Config_IICA*n*}_Slave_Receive function.

[Syntax]

```
void    R_{Config_IICAn}_slave_handler(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_callback_slave_sendend

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

Remark 1. This API function is called as the callback routine of interrupt process R_{Config_IICAn}_slave_handler.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void     r_{Config_IICA n}_callback_slave_sendend(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_callback_slave_receiveend

This API function executes processing in response to slave receive completed.

Remark 1. This API function is called as the callback routine of interrupt process R_{Config_IICAn}_slave_handler.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void     r_{Config_IICA n}_callback_slave_receiveend(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_IICA*n*}_callback_slave_error

This API function executes processing in response to the detection of an addresses not match or NACK error.

Remark 1. This API function is called as the callback routine of interrupt process R_{Config_IICAn}_slave_handler.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void    r_{Config_IICAn}_callback_slave_error(MD_STATUS flag);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
I	MD_STATUS flag;	Status flag MD_NACK: NACK error MD_ERROR: addresses not match error

[Return value]

None.

r_{Config_IICA*n*}_callback_getstopcondition

This API function executes processing in response to IICA*n* get a slave stop condition.

Remark 1. This API function is called as the callback routine of interrupt process R_{Config_IICAn}_slave_handler.

Remark 2. Please take note to keep necessary flag set/clear in callback function and move other processing code out of callback and interrupt function. Otherwise, the next interrupt won't be processed at the correct timing.

[Syntax]

```
static void    r_{Config_IICAn}_callback_getstopcondition(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

Refer to Serial Array Unit IIC master mode Usage example.

4.2.29 UART Communication (Serial Interface UARTA)

Below is a list of API functions output by the Smart Configurator for UART communication (for serial interface UARTA) use.

Table 4-29 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_UARTA*n*}_Create	Serial Interface UARTA	Executes initialization processing that is required before controlling the UARTA*n* module.
R_{Config_UARTA*n*}_Start		Starts UARTA*n* module operation.
R_{Config_UARTA*n*}_Stop		Stops UARTA*n* module operation.
R_{Config_UARTA*n*}_Send		Sends UARTA*n* data.
R_{Config_UARTA*n*}_Receive		Receives UARTA*n* data.
R_{Config_UARTA*n*}_Loopback_Enable		Enables the UARTA*n* loopback function.
R_{Config_UARTA*n*}_Loopback_Disable		Disables the UARTA*n* loopback function.
R_{Config_UARTA*n*}_Create_UserInit		Executes user-specific initialization processing for the UARTA*n*.
R_{Config_UARTA*n*}_PollingEnd_UserCode		Executes user code in response to completion of continuous transmission by polling.
r_{Config_UARTA*n*}_interrupt_send		Executes processing in response to UARTA*n* transmission completion interrupt.
r_{Config_UARTA*n*}_interrupt_receive		Executes processing in response to UARTA*n* reception transfer end interrupt.
r_{Config_UARTA*n*}_interrupt_error		Executes processing in response to UARTA*n* reception communication error interrupt.
r_{Config_UARTA*n*}_callback_sendend		Executes processing in response to UARTA*n* transmission completion interrupt.
r_{Config_UARTA*n*}_callback_receiveend		Executes processing in response to UARTA*n* reception transfer end interrupt.
r_{Config_UARTA*n*}_callback_error		Executes processing in response to UARTA*n* reception communication error interrupt.

R_{Config_UARTA*n*}_Create

This API function executes initialization processing that is required before controlling the UARTA*n* module.

Remark This API function is called from R_UARTA_Create.

[Syntax]

```
void R_{Config_UARTAn}_Create(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UARTA*n*}_Start

This API function starts UARTA*n* module operation.

[Syntax]

```
void    R_{Config_UARTAn}_Start(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UARTA*n*}_Stop

This API function stops UARTA*n* module operation.

[Syntax]

```
void    R_{Config_UARTAn}_Stop(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UARTA*n*}_Send

This API function sends UARTA*n* data.

[Syntax]

```
MD_STATUS      R_{Config_UARTAn}_Send(uint8_t * const tx_buf, uint16_t tx_num);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t * const tx_buf;	Transfer buffer pointer
I	uint16_t tx_num;	Buffer size

[Return value]

Macro	Description
MD_OK	Normal end (send the first data)
MD_ARGERROR	Error argument input error.

R_{Config_UARTA*n*}_Receive

This API function receives UARTA*n* data.

[Syntax]

```
MD_STATUS      R_{Config_UARTAn}_Receive(uint8_t * const rx_buf, uint16_t rx_num);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
O	uint8_t * const rx_buf;	Receive buffer pointer
I	uint16_t tx_num;	Buffer size

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_UARTA*n*}_Loopback_Enable

This API function enables the UARTA*n* loopback function.

[Syntax]

```
void    R_{Config_UARTAn}_Loopback_Enable(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UARTA*n*}_Loopback_Disable

This API function disables the UARTA*n* loopback function.

[Syntax]

```
void    R_{Config_UARTAn}_Loopback_Disable(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UARTA*n*}_Create_UserInit

This API function executes user-specific initialization processing for the UARTA*n*.

Remark This API functions is called from R_{Config_UARTA*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_UARTAn}_Create_UserInit(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_UARTA*n*}_PollingEnd_UserCode

This API function executes user code in response to completion of continuous transmission by polling.

Remark This API function is called from R_{Config_UARTA*n*}_Send corresponding to data transmission completion.

.

[Syntax]

```
void    R_{Config_UARTAn}_PollingEnd_UserCode(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UARTA*n*}_interrupt_send

This API function executes processing in response to UARTA*n* transmission completion interrupt.

[Syntax]

```
void    r_{Config_UARTAn}_interrupt_send(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UARTA*n*}_interrupt_receive

This API function executes processing in response to UARTA*n* reception transfer end interrupt.

[Syntax]

```
void    r_{Config_UARTAn}_interrupt_receive(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UARTA*n*}_interrupt_error

This API function executes processing in response to UARTA*n* reception communication error interrupt.

[Syntax]

```
void    r_{Config_UARTAn}_interrupt_error(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UARTA*n*}_callback_sendend

This API function executes processing in response to UARTA*n* transmission completion interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_UARTA*n*}_interrupt_send corresponding to the UARTA*n* transmission completion interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_UARTAn}_callback_sendend(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UARTA*n*}_callback_receiveend

This API function executes processing in response to UARTA*n* reception transfer end interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_UARTA*n*}_interrupt_receive corresponding to the UARTA*n* reception transfer end interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_UARTAn}_callback_receiveend(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_UARTA*n*}_callback_error

This API function executes processing in response to UARTA*n* reception communication error interrupt.

Remark 1. This API function is called as the callback routine of interrupt process r_{Config_UARTA*n*}_interrupt_error corresponding to the UARTA*n* reception communication error interrupt.

Remark 2. User should only keep necessary flag set/clear in callback function, other processing code should be moved out of callback and interrupt function. Otherwise, the interrupt is not processed at the correct timing.

[Syntax]

```
static void     r_{Config_UARTAn}_callback_error(uint8_t err_type);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t err_type;	Error type value: Bit0: Overrun error Bit1: Framing error Bit2: Parity error Bit3 to Bit7: 0

[Return value]

None.

Usage example

This is an example for UARTA0 transmit and receieve data by polling mode and UARTA1 also transmit and receieve data twice:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf0[] = {0x7A};
uint8_t tx_buf1[] = {0x7A, 0x6C, 0x27, 0x1F, 0xF8};
uint8_t rx_buf0[] = {0x00};
uint8_t rx_buf1[] = {0x00, 0x00, 0x00, 0x00, 0x00};
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receptend_flag = 0U;
int main(void);

int main(void)
{
    R_Config_UARTA0_Start();
    R_Config_UARTA1_Start();

    // 1st transmission (UARTA0 to UARTA1)
    R_Config_UARTA1_Receive(rx_buf0, sizeof(rx_buf0));
    R_Config_UARTA0_Send(tx_buf0, sizeof(tx_buf0));
    while((1U != transmitend_flag) || (1U != receptend_flag));

    // 2nd transmission (UARTA1 to UARTA0)
    R_Config_UARTA0_Receive(rx_buf1, sizeof(rx_buf1));
    R_Config_UARTA1_Send(tx_buf1, sizeof(tx_buf1));
    while((2U != transmitend_flag) || (2U != receptend_flag));

    R_Config_UARTA0_Stop();
    R_Config_UARTA1_Stop();

    return 0;
}
```

Config_UARTA0_user.c

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

void R_Config_UARTA0_PollingEnd_UserCode(void)
{
    /* Start user code for R_Config_UARTA0_PollingEnd_UserCode. Do not edit comment generated here */
    transmitend_flag++;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_UARTA0_callback_receiveend (void)
{
    /* Start user code for r_Config_UARTA0_callback_sendend. Do not edit comment generated here */
    receptend_flag++;
    /* End user code. Do not edit comment generated here */
}
```

Config_UARTA1_user.c

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

void R_Config_UARTA1_PollingEnd_UserCode (void)
{
    /* Start user code for R_Config_UARTA1_PollingEnd_UserCode. Do not edit comment generated here */
    transmitend_flag++;
    /* End user code. Do not edit comment generated here */
}

static void r_Config_UARTA1_callback_receiveend (void)
{
    /* Start user code for r_Config_UARTA1_callback_receiveend. Do not edit comment generated here */
    receptend_flag++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.30 Remote Control Signal Receiver

Below is a list of API functions output by the Smart Configurator for remote control signal receiver use.

Table 4-30 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_REMC}_Create	Remote Control Signal Receiver	Executes initialization processing that is required before controlling the REMC module.
R_{Config_REMC}_Start		Starts operation of the remote control signal receiver.
R_{Config_REMC}_Stop		Stops operation of the remote control signal receiver.
R_{Config_REMC}_Read		Specifies the location where the received data are to be stored and the number of bytes to be received.
R_{Config_REMC}_Create_UserInit		Executes user-specific initialization processing for the the remote control signal receiver.
r_{Config_REMC}_interrupt		Executes processing in response to REMC interrupt.
r_{Config_REMC}_callback_receiveend		Executes processing in response to data reception complete interrupts.
r_{Config_REMC}_callback_comparematch		Executes processing in response to compare match interrupts.
r_{Config_REMC}_callback_receiveerror		Executes processing in response to receive error interrupt.
r_{Config_REMC}_callback_bufferfull		Executes processing in response to receive buffer full interrupts.
r_{Config_REMC}_callback_header		Executes processing in response to header pattern match interrupt.
r_{Config_REMC}_callback_data0		Executes processing in response to data "0" pattern match interrupt.
r_{Config_REMC}_callback_data1		Executes processing in response to data "1" pattern match interrupt.
r_{Config_REMC}_callback_specialdata		Executes processing in response to special data pattern match interrupt.

R_{Config_REMC}_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_REMC}_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_REMC}_Start

This API function starts operation of the remote control signal receiver.

[Syntax]

```
void    R_{Config_REMC}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_REMC}_Stop

This API function stops operation of the remote control signal receiver.

[Syntax]

```
void    R_{Config_REMC}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_REMC}_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_REMC}_Read(uint8_t * const rx_buf, uint8_t rx_num);
```

[Argument(s)]

I/O	Argument(s)	Description
O	uint8_t * const rx_buf;	Pointer to the buffer where the received data are to be stored
O	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_REMC}_Create_UserInit

This API function executes user-specific initialization processing for the the remote control signal receiver.

Remark This API functions is called from R_{Config_REMC}_Create as a callback routine.

[Syntax]

```
void    R_{Config_REMC}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_interrupt

This API function executes processing in response to REMC interrupt.

[Syntax]

```
void    r_{Config_REMC}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_callback_receiveend

This API function executes processing in response to data reception complete interrupts.

Remark This API function is called as the callback routine of interrupt process r_{Config_REMC}_interrupt corresponding to data reception completion.

[Syntax]

```
static void r_{Config_REMC}_callback_receiveend(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_callback_comparematch

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

Remark This API function is called as the callback routine of interrupt process r_{Config_REMC}_interrupt corresponding to the REMC compare match is triggered.

[Syntax]

```
static void r_{Config_REMC}_callback_comparematch(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_callback_receiveerror

This API function executes processing in response to receive error interrupt.

Remark This API function is called as the callback routine of interrupt process r_{Config_REMC}_interrupt corresponding to receive error.

[Syntax]

```
static void     r_{Config_REMC}_callback_receiveerror(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_callback_bufferfull

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

Remark This API function is called as the callback routine of interrupt process r_{Config_REMC}_interrupt corresponding to the REMC receive buffer full.

[Syntax]

```
static void     r_{Config_REMC}_callback_bufferfull(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_callback_header

This API function executes processing in response to header pattern match interrupt.

Remark This API function is called as the callback routine of interrupt process r_{Config_REMC}_interrupt corresponding to the REMC header pattern match.

[Syntax]

```
static void r_{Config_REMC}_callback_header(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_callback_data0

This API function executes processing in response to data "0" pattern match interrupt.

Remark This API function is called as the callback routine of interrupt process r_{Config_REMC}_interrupt corresponding to the REMC data "0" pattern match.

[Syntax]

```
static void     r_{Config_REMC}_callback_data0(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_callback_data1

This API function executes processing in response to data "1" pattern match interrupt.

Remark This API function is called as the callback routine of interrupt process r_{Config_REMC}_interrupt corresponding to the REMC data "1" pattern match.

[Syntax]

```
static void r_{Config_REMC}_callback_data1(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_REMC}_callback_specialdata

This API function executes processing in response to special data pattern match interrupt.

Remark This API function is called as the callback routine of interrupt process r_{Config_REMC}_interrupt corresponding to the REMC special data pattern match.

[Syntax]

```
static void     r_{Config_REMC}_callback_specialdata(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for stopping operation of the remote control signal receiver at the end of data reception:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

volatile uint8_t g_remc_rx_buf[8];

int main(void);

int main(void)
{
    /* Start the REMC operation */
    R_Config_REMC_Start();

    /* Read data from receive data buffer */
    R_Config_REMC_Read((uint8_t *)g_remc_rx_buf, 8U);

    return 0;
}
```

Config_REMC_user.c

```
static void r_Config_REMC_callback_receiveend(void)
{
    /* Start user code for r_Config_REMC_callback_receiveend. Do not edit comment generated here */
    R_Config_REMC_Stop();
    /* End user code. Do not edit comment generated here */
}
```

4.2.31 A/D Converter

Below is a list of API functions output by the Smart Configurator for A/D converter use.

Table 4-31 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_ADC}_Create	A/D Converter	Executes initialization processing that is required before controlling the ADC module.
R_{Config_ADC}_Start		Starts the AD converter.
R_{Config_ADC}_Stop		Stops the AD converter.
R_{Config_ADC}_Set_OperationOn		Enables AD voltage comparator operation.
R_{Config_ADC}_Set_OperationOff		Disables AD voltage comparator operation.
R_{Config_ADC}_Set_ADChannel		Selects analog input channel.
R_{Config_ADC}_Set_SnoozeOn		Enables AD wakeup function.
R_{Config_ADC}_Set_SnoozeOff		Disables AD wakeup function.
R_{Config_ADC}_Set_TestChannel		Sets test function.
R_{Config_ADC}_Get_Result_10bit		Returns the high 10 bits conversion result in the buffer.
R_{Config_ADC}_Get_Result_8bit		Returns the high 8 bits conversion result in the buffer.
R_{Config_ADC}_Get_Result_12bit		Returns the low 12 bits conversion result in the buffer.
R_{Config_ADC}_Create_UserInit		Executes user-specific initialization processing for the AD converter.
r_{Config_ADC}_interrupt		Executes processing in response to A/D conversion end interrupt.

R_{Config_ADC}_Create

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

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

[Syntax]

```
void    R_{Config_ADC}_Create(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ADC}_Start

This API function starts the AD converter.

[Syntax]

```
void    R_{Config_ADC}_Start(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ADC}_Stop

This API function stops the AD converter.

[Syntax]

```
void    R_{Config_ADC}_Stop(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ADC}_Set_OperationOn

Enables AD voltage comparator operation.

[Syntax]

```
void    R_{Config_ADC}_Set_OperationOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ADC}_Set_OperationOff

Disables AD voltage comparator operation.

[Syntax]

```
void    R_{Config_ADC}_Set_OperationOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ADC}_Set_ADChannel

This API function selects analog input channel.

[Syntax]

```
MD_STATUS      R_{Config_ADC}_Set_ADChannel(e_ad_channel_t channel);
```

[Argument(s)]

I/O	Argument(s)	Description
I	e_ad_channel_t channel;	Analog input channel

Remark Below is shown an example of the structure e_ad_channel_t channel (channel conditions).

```
typedef enum
{
    ADCHANNEL0, ADCHANNEL1, ADCHANNEL2, ADCHANNEL3, ADCHANNEL4,
    ADCHANNEL5, ADCHANNEL6, ADCHANNEL7, ADCHANNEL8, ADCHANNEL9,
    ADCHANNEL10, ADCHANNEL11, ADCHANNEL12, ADCHANNEL13,
    ADCHANNEL14, ADCHANNEL16 = 16U, ADCHANNEL17, ADCHANNEL18,
    ADCHANNEL19, ADCHANNEL20, ADCHANNEL21, ADCHANNEL22,
    ADCHANNEL23, ADCHANNEL24, ADCHANNEL25, ADCHANNEL26,
    ADTEMPERSENSOR0 = 128U, ADINTERREFVOLT
} e_ad_channel_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_ADC}_Set_SnoozeOn

Enables AD wakeup function.

[Syntax]

```
void    R_{Config_ADC}_Set_SnoozeOn(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ADC}_Set_SnoozeOff

Disables AD wakeup function.

[Syntax]

```
void    R_{Config_ADC}_Set_SnoozeOff(void);
```

[Argument(s)]

None.

[Return value]

None.

R_{Config_ADC}_Set_TestChannel

This API function sets test function.

[Syntax]

```
MD_STATUS      R_{Config_ADC}_Set_TestChannel(e_test_channel_t channel);
```

[Argument(s)]

I/O	Argument(s)	Description
I	e_test_channel_t channel;	Sets test channel

Remark Below is shown the structure e_test_channel_t channel (input channel conditions).

```
typedef enum
{
    ADNORMALINPUT,
    ADAVREFM = 2U,
    ADAVREFP
} e_test_channel_t;
```

[Return value]

Macro	Description
MD_OK	Normal end
MD_ARGERROR	Error argument input error.

R_{Config_ADC}_Get_Result_10bit

This API function returns the high 10 bits conversion result in the buffer.

[Syntax]

```
void    R_{Config_ADC}_Get_Result_10bit(uint16_t * const buffer);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t * const buffer;	The address where to write the conversion result

[Return value]

None.

R_{Config_ADC}_Get_Result_8bit

This API function returns the high 8 bits conversion result in the buffer.

[Syntax]

```
void    R_{Config_ADC}_Get_Result_8bit(uint8_t * const buffer);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t * const buffer;	The address where to write the conversion result

[Return value]

None.

R_{Config_ADC}_Get_Result_12bit

This API function returns the low 12 bits conversion result in the buffer.

[Syntax]

```
void    R_{Config_ADC}_Get_Result_12bit(uint16_t * const buffer);
```

[Argument(s)]

I/O	Argument(s)	Description
I	uint16_t * const buffer;	the address where to write the conversion result

[Return value]

None.

R_{Config_ADC}_Create_UserInit

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

Remark This API functions is called from R_{Config_ADC}_Create as a callback routine.

[Syntax]

```
void    R_{Config_ADC}_Create_UserInit(void);
```

[Argument(s)]

None.

[Return value]

None.

r_{Config_ADC}_interrupt

This API function executes processing in response to A/D conversion end interrupt.

[Syntax]

```
void    r_{Config_ADC}_interrupt(void);
```

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for getting the 8-bit A/D conversion result:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
uint8_t   adc_data[1] = {0};
extern uint8_t   adc_Interrupt_flag;

int main(void);

int main(void)
{
      adc_Interrupt_flag = 0U;
      /* Start comparator 0 */
      R_Config_ADC_Set_OperationOn();
      R_Config_ADC_Start ();
      while(adc_Interrupt_flag != 1U);
      R_Config_ADC_Get_Result_8bit(adc_data);
      R_Config_ADC_Stop ();
      R_Config_ADC_Set_OperationOff();

      return 0;
}
```

Config_ADC_user.c

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

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

4.2.32 D/A Converter

Below is a list of API functions output by the Smart Configurator for D/A converter use.

Table 4-32 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_DAC*n*}_Create	D/A Converter	Executes initialization processing that is required before controlling the DAC*n* module.
R_{Config_DAC*n*}_Start		Starts the DAC*n* module.
R_{Config_DAC*n*}_Stop		Stops the DAC*n* module.
R_{Config_DAC*n*}_Set_ConversionValue		Sets the DAC*n* value to convert.
R_{Config_DAC*n*}_Create_UserInit		Executes user-specific initialization processing for the DAC*n*.

R_{Config_DAC*n*}_Create

This API function executes initialization processing that is required before controlling the DAC*n* module.

Remark This API function is called from R_DAC_Create.

[Syntax]

```
void R_{Config_DACn}_Create(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_DAC*n*}_Start

This API function starts the DAC*n* converter.

[Syntax]

```
void    R_{Config_DACn}_Start(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_DAC*n*}_Stop

This API function stops the DAC*n* converter.

[Syntax]

```
void    R_{Config_DACn}_Stop(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_DAC*n*}_Set_ConversionValue

This API function sets the DAC*n* value to convert.

[Syntax]

```
void    R_{Config_DACn}_Set_ConversionValue(uint8_t reg_value);
```

Remark *n* is the channel number.

[Argument(s)]

I/O	Argument(s)	Description
I	uint8_t reg_value;	Value of conversion

[Return value]

None.

R_{Config_DAC*n*}_Create_UserInit

This API function executes user-specific initialization processing for the DAC*n*.

Remark This API functions is called from R_{Config_DAC*n*}_Create as a callback routine.

[Syntax]

```
void    R_{Config_DAC n}_Create_UserInit(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for starting D/A conversion with a user-define value:

(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"

int main(void);

int main(void)
{
    R_Config_DAC0_Set_ConversionValue(0xF0);
    R_Config_DAC0_Start();

    return 0;
}
```

4.2.33 Comparator

Below is a list of API functions output by the Smart Configurator for comparator use.

Table 4-33 API Functions:

API Function Name	Peripheral Name	Description
R_{Config_COMP*n*}_Create	Comparator	Executes initialization processing that is required before controlling the comparator *n* module.
R_{Config_COMP*n*}_Start		Starts the comparator *n*.
R_{Config_COMP*n*}_Stop		Stops the comparator *n*.
R_{Config_COMP*n*}_Create_UserInit		Executes user-specific initialization processing for the comparator *n*.
r_{Config_COMP*n*}_interrupt		Executes processing in response to comparator detection interrupt.

R_{Config_COMP*n*}_Create

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

Remark This API function is called from R_COMP_Create.

[Syntax]

```
void    R_{Config_COMPn}_Create(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_COMP*n*}_Start

This API function starts the comparator n converter.

[Syntax]

```
void    R_{Config_COMPn}_Start(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_COMP*n*}_Stop

This API function stops the comparator n converter.

[Syntax]

```
void    R_{Config_COMPn}_Stop(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

R_{Config_COMP*n*}_Create_UserInit

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

Remark This API functions is called from R_{Config_COMP*n*}_Create as a callback routine.

[Syntax]

```
void R_{Config_COMPn}_Create_UserInit(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

r_{Config_COMP*n*}_interrupt

This API function executes processing in response to comparator detection interrupt.

Remark This API function is called as the interrupt handler for comparator detection interrupt, which occur when an active edge of the comparator output is detected.

[Syntax]

```
void    r_{Config_COMPn}_interrupt(void);
```

Remark *n* is the channel number.

[Argument(s)]

None.

[Return value]

None.

Usage example

This is an example for setting a flag when detecting an active edge of the comparator output:
(Blue code is user code.)

main.c

```
#include "r_smc_entry.h"
extern uint8_t comp0_trig_flag;

int main(void);

int main(void)
{
    comp0_trig_flag = 0U;

    /* Start comparator 0 */
    R_Config_COMP0_Start ();
    while(comp0_trig_flag != 1U);
    comp1_trig_flag = 0U;
    R_Config_COMP0_Stop ();

    return 0;
}
```

Config_COMP0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t comp0_trig_flag = 0U;
/* End user code. Do not edit comment generated here */

static void __near r_Config_COMP0_interrupt(void)
{
    /* Start user code for r_Config_COMP0_interrupt. Do not edit comment generated here */
    /* Set the flag */
    comp0_trig_flag = 1U;
    /* End user code. Do not edit comment generated here */
}
```

Revision Record

Rev.	Section	Description
1.00	–	First Edition issued

Smart Configurator User's Manual: RISC-V MCU API Reference

Publication Date: Rev.1.00 Jan 22, 2024

Published by: Renesas Electronics Corporation

Smart Configurator

Renesas Electronics Corporation

R20UT5385EC0100